

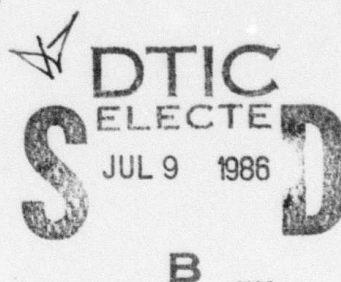
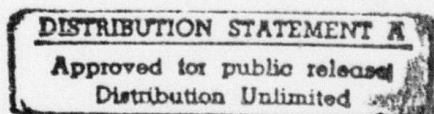
ADVANCED TELEPROCESSING SYSTEMS
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

SEMI-ANNUAL TECHNICAL REPORT

March 31, 1986

Principal Investigator: Leonard Kleinrock

Computer Science Department
School of Engineering and Applied Science
University of California
Los Angeles



86 7 7 05 8

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|--|-------------------------------------|--|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. ADA 169696 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) Advanced Teleprocessing Systems: Semi-Annual Technical Report | | 5. TYPE OF REPORT & PERIOD COVERED Semi-Annual Technical 10/1/85 - 3/31/86 |
| 7. AUTHOR(s) Leonard Kleinrock | | 6. PERFORMING ORG. REPORT NUMBER |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Engineering & Applied Science University of California, Los Angeles Los Angeles, Ca 90024 | | 8. CONTRACT OR GRANT NUMBER(s) MDA 903-82-C-0064 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS DARPA Order No. 2496 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | | 12. REPORT DATE March 31, 1986 |
| | | 13. NUMBER OF PAGES 50 |
| | | 15. SECURITY CLASS. (of this report) |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |
| 16. DISTRIBUTION STATEMENT (of this Report) Approved for Public Release; Distribution Unlimited. | | |
| 17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) | | |
| 18. SUPPLEMENTARY NOTES | | |
| 19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Random access communications, computer networks, distributed processing, distributed algorithms for election and traversal in networks, parallel processing systems. | | |
| 20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This semi-annual technical report covers research carried out by the Advanced Teleprocessing Systems Group at UCLA under DARPA Contract no. MDA 903-82-C-0064 covering the period from October 1, 1985 to March 31, 1986. | | |

This Semi - Annual Technical Report covers research carried out by the Advanced Teleprocessing Systems Group at UCLA under DARPA Contract No. MDA 903-82-C-0064 covering the period from October 1, 1985 to March 31, 1986.

In this six-month period, six papers were published in the professional literature. These papers were in the fields of computer networks, multiaccess communications, and distributed processing. Our main focus is currently in the direction of distributed systems performance; we reproduce two papers in that area as the main body of this report. These papers are "Distributed Systems," by Leonard Kleinrock, and "Broadcast Communications and Distributed Algorithms," by Rina Dechter and Leonard Kleinrock.

QUALITY
INSPECTED

1

| | |
|--------------------|-------------------------------------|
| Accession For | |
| NTIS GRA&I | <input checked="" type="checkbox"/> |
| DTIC TAB | <input type="checkbox"/> |
| Unannounced | <input type="checkbox"/> |
| Justification | |
| By | |
| Distribution/ | |
| Availability Codes | |
| Dist | Avail and/or Special |
| A-1 | |

DTIC
ELECTE
JUL 9 1986
S B D

ADVANCED TELEPROCESSING SYSTEMS

Semi-Annual Technical Report

March 31, 1986

Contract Number: MDA 903-82-C-0064

DARPA Order Number: 2496

Contract Period: February 1, 1984 to June 30, 1986

Report Period: October 1, 1985 to March 31, 1986

Principal Investigator: Leonard Kleinrock

Co-Principal Investigator: Mario Gerla

(213) 825-2543

Computer Science Department
School of Engineering and Applied Science
University of California, Los Angeles

Sponsored by

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

ADVANCED TELEPROCESSING SYSTEMS

Defense Advanced Research Projects Agency
Semi - Annual Technical Report

March 31, 1986

INTRODUCTION

This Semi - Annual Technical Report covers research carried out by the Advanced Teleprocessing Systems Group at UCLA under DARPA Contract No. MDA 903-82-C-0064 covering the period from October 1, 1985 to March 31, 1986. Under this contract we have three designated tasks as follows:

TASK I. DISTRIBUTED COMMUNICATIONS ACCESS

The general problem of sharing a multi-access broadcast distributed systems among a set of competing users will be studied. General issues involving exhaustive communications, start-up problems and refined models to manifest some more realistic phenomena in these systems will be studied. Applications to packet radio systems and large survivable networks involving the study of tandem networks, multi-hop networks, one-way communication links, correct reception of more than one simultaneous transmission and mobility will be included. Further applications will include the study of very high bandwidth channels and/or very long propagation delay systems, multiple token systems and compound hierarchical network structures.

TASK II. DISTRIBUTED PROCESSING

The interplay between distributed communications in a broadcast environment and processing of distributed data will be studied. For example, the effect of merging sorted lists in a broadcast environment, as well as finding properties of elements in these lists, will be studied. Concurrency in multiprocessor systems will be studied in order to investigate performance in terms of response time and speedup factors for various graph models of computation. Connection architectures for multiprocessor systems will be investigated as well. One application here is the structure of the processing and communication architecture for supercomputers.

TASK III. DISTRIBUTED CONTROL AND ALGORITHMS

Routing, flow control and survivability in large packet radio networks as well as in public data networks will be studied as control algorithms in a distributed environment. Measures of performance, including throughput, response time, blocking, power, fairness, and robustness will be applied to these systems. Distributed algorithms for finding shortest paths, connectivity, loops, etc. will be studied. The effect of node and link failures, limited amounts of memory at each node and restricted channel capacity for communications will be investigated. The effect of network failures and delays on distributed data base management systems will also be studied.

In this six-month period, six papers were published in the professional literature. These papers were in the fields of computer networks, multiaccess communications, and distributed processing. Our main focus is currently in the direction of distributed systems performance; we reproduce two papers in that area as the main body of this report. These papers are "Distributed Systems," by Leonard Kleinrock, and "Broadcast Communications and Distributed Algorithms," by Rina Dechter and Leonard Kleinrock.

RESEARCH PUBLICATIONS

1. Kleinrock, L., "Distributed Systems," invited paper for *ACM/IEEE-CS Joint Special Issue (November 1985): Communications of the ACM*, Vol. 28, No. 11, pp. 1200-1213, and *Computer*, Vol. 18, No. 11, pp. 90-103.

Growth of distributed systems has attained unstoppable momentum. If we better understood how to think about, analyze, and design distributed systems, we could direct their implementation with more confidence.

2. Rodrigues, P., Fratta, L., and M. Gerla, "Tokenless Protocols for Fiber Optic Local Area Networks," *IEEE Journal on Selected Areas in Communications*, Vol. SAC-3, No. 6, November 1985, pp. 928-940.

A family of LAN (Local Area Network) protocols is presented. The LAN consists of a pair of unidirectional fiber optic buses to which stations are connected via passive taps. The protocols provide round-robin bounded delay access to all stations. Contrary to most round-robin access schemes, the protocols do not require transmission of special packets (tokens); rather, they simply rely on the detection of bus activity at each station. The performance of these protocols in various traffic conditions and system configurations is evaluated via analysis and simulation.

3. Dechter, R. and L. Kleinrock, "Broadcast Communications and Distributed Algorithms," *IEEE Transactions on Computers*, Vol. C-35, No. 3, March 1986, pp 210-219.

The paper addresses ways in which one can use "broadcast communication" in distributed algorithms and the relevant issues of design and complexity. We present an algorithm for merging k sorted lists of n/k elements using k processors and prove its worst case complexity to be $2n$, regardless of the number of processors, while neglecting the cost arising from possible conflicts on the broadcast channel. We also show that this algorithm is optimal under single-channel broadcast communications. In a variation of the algorithm, we show that by using an extra local memory of $O(k)$ the number of broadcasts is reduced to n . When the algorithm is used for sorting n elements with k processors, where each processor sorts its own list first and then merging, it has a complexity of $O(n/k \log(n/k) + n)$, and is thus asymptotically optimal for large n . We also discuss the cost incurred by the channel access scheme and prove that resolving conflicts whenever k processors are involved introduces a cost factor of at least $\log k$.

This report consists of two reprints:

① DISTRIBUTED SYSTEMS:

Growth of distributed systems has attained unstoppable momentum. If we better understood how to think about, analyze, and design distributed systems, we could direct their implementation with more confidence.

LEONARD KLEINROCK

→ top 1 of 2d paper →

DISTRIBUTED SYSTEMS IN NATURE

How did the killer bees find their way up to North America? By what mechanism does a colony of ants carry out its complex tasks? What guides and controls a flock of birds or a school of fish? The answers to these questions involve examples of loosely coupled systems that achieve a common goal with distributed control.

Throughout nature we find an enormous amount of processing taking place at the level of the individual organism (be it an ant, a sparrow, or a human), and we have only begun to comprehend how processing and memory functions operate, especially in the human species. How does a human perform the acts of perception, cognition, decision making, and motor control? This processing occurs in a fraction of a second, using natural processing elements that are orders of magnitude slower than our current computer processing elements [8].

We do know that the brain is organized and structured very differently from our present computing machines. In human beings (i.e., in their internal neural systems) and in groups of organisms, nature has been extremely successful in implementing distributed systems that are far more clever and impressive than any computing machine humans have yet devised. We have succeeded in manufacturing highly complex devices capable of high-speed computation and massive accurate memory, but we have not yet gained sufficient understanding of distributed systems—our systems are still highly constrained and rigid in their construction and behavior. The gap between natural and man-made systems is huge, and we have a long way to go before

we bridge the gap in understanding and implementation (see Figure 1, pp. 1202–1203).

WHY SHOULD WE STUDY DISTRIBUTED SYSTEMS?

Currently we are experiencing the effects of the confluence of powerful forces in information technology. By far, the most significant effect is the host of revolutionary changes that have been brought about by the integrated chip—especially in the form of VLSI and the resulting enormous improvements in processing, storage, and communications. At the same time, we are experiencing a frightening backlog in software-application development while the user community is clamoring for unprecedented power in processing, communications, storage, and applications. Fortunately, we have the potential for this power—if only we could figure out how to put all the pieces together!

Distributed systems have come into existence in our industrial society in some very natural ways. For example, we have seen the emergence of a large number of distributed databases—systems that have evolved because the source of the data is not centralized and where there is a local need for frequent and immediate access to the locally generated data (e.g., the employee database at a branch office of a nationwide organization) in addition to a global need to view the entire database. Situations such as these require us to place some processing power at the many distributed locations for collecting, preprocessing, and accessing data. On-line transaction processing is an application that may contain a local component as well as a distributed-processing component, and the current proliferation of desktop personal computers is a manifestation of distributed-processing power. Indeed, if we measure

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-82-C0064.

ACM/IEEE-CS Joint Issue © 1985 ACM 0001-0782/85/1100-1200 75¢

processing power in MIPS (millions of instructions per second), we note that the number of installed MIPS in personal computers is an order of magnitude greater than the number installed in mainframes. However, most of those PC MIPS lie idle most of the time. Imagine what a terrific distributed-processing system we could fire up with that unused power! When data and processing are distributed, we are obliged to provide communications to link the resources. Thus we are led into the use of packet networks, satellite networks, internets, cellular and packet radio networks, metropolitan area networks, and local area networks.

Distributed systems can provide the necessary power to meet the growing demands of the user community. We are demanding capability faster than the advances in devices alone can supply, and to meet these demands we will have to rely on innovative computing architectures such as parallel-processing systems. These large distributed databases, along with distributed-processing and distributed-communication networks, have given rise to some very complex distributed-system structures, and it is essential that we learn how to think about them properly (see Figure 2, pp. 1204-1205).

ARCHITECTURE AND ALGORITHMS

The world of applications has an insatiable need for computing power. A good mathematician can easily consume any finite computing capability by posing a combinatoric problem whose computational complexity grows exponentially with a variable of the problem (e.g., the enumeration of all graphs with N nodes). The ways in which we push back this "power wall" involve both hardware and software solutions. Typically, the methods for speeding up the computation include the following:

- faster devices (a physics and engineering problem),
- architectures that permit concurrent processing (a system design problem),
- optimizing compilers for detecting concurrency (a software-engineering problem),
- algorithms for specification of concurrency (a language problem), and
- more expressive models of computation (an analytic problem).

Characterizing the Architecture

There are many ways of classifying machine architectures—too many, in fact. The following classification was selected for the purposes of this article.

We begin with the purely serial uniprocessor in which a single instruction stream operates on a single data stream (SISD). These systems are "centralized" at the global level, but really do contain many elements of a distributed system at the lower levels, for example, at the level of communications on the VLSI chips themselves.

Next is the vector machine, in which a single in-

struction stream operates on a multiple data stream (SIMD). These include array processors (e.g., systolic arrays) and pipeline processors.

The third consists of multiple processors that, collectively, can process multiple instruction streams on multiple data streams (MIMD). The form of multiprocessing that takes place when multiple processors cooperate closely to process tasks from the same job is referred to as parallel processing. On the other hand, the term distributed processing is applied to the form of multiprocessing that takes place when the multiple processors cooperate loosely and process separate jobs.

Vector machines and multiprocessing systems all provide some form of concurrency. The effect of this concurrency on system performance is important and is therefore a very active area of research (see Figure 3, p. 1206).

Since the onslaught of the VLSI revolution, a number of machine architectures have been implemented in an attempt to provide the supercomputing power toward which concurrent processing tempts us [5]. Two excellent recent summaries of some of these projects are offered by Hwang [9] and by Schneck et al. [17]. There you will find the Butterfly machine, the Cosmic Cube, various kinds of tree machines, the Cedar project, the Sisal language, the Connection machine, and others whose names are intriguingly close to Mother Nature's systems.

Characterizing the Algorithm

The major goal in characterizing the algorithm is to identify and exploit its inherent parallelism (i.e., potential for concurrency). The levels of resolution at which we can attempt to find this parallelism are listed below in decreasing order of granularity [16]:

- job execution,
- task execution,
- process execution,
- instruction execution,
- register transfer, and
- logic device.

Clearly, as we drop down the list to finer granularity, we expose more and more parallelism, but we also increase the complexity of scheduling these tiny objects to the processors and of providing the communications among so many objects (the problem of interprocess communication—IPC). As was stated earlier, if we operate at the top level (i.e., at the job level), then we think of the system as a distributed-processing system; if we operate at the task or process level, we have a parallel-processing system; if we operate at the instruction level, we have the vector machine and the array processor.

Regardless of the level at which we operate, it behooves us to create a "model" of the algorithm or, if you will, of the computation we are processing [10]. A very common model is the graph model of computation, which is normally used at the task or process level

(another common modeling method is the use of Petri Nets). In this model, the nodes represent the tasks (or processes), and the directed edges represent the dependencies among the tasks, thereby displaying the partial ordering of the tasks and the parallelism that can be exploited (see Figure 4, p. 1207).

However, the problem of finding the parallelism in the lines of code that represent the algorithm still remains, and there is an ongoing effort to simplify (and even automate) this task by developing parallel programming languages for implementing these algorithms (e.g., Ada[®], concurrent Pascal).

Matching the Architecture to the Algorithm

The performance of a distributed system depends strongly on how well the architecture and the algorithm are matched. For example, a highly parallel algorithm will perform well on a highly parallel architecture; a distributed system requiring lots of interprocessor communication will perform poorly if the communication bandwidth is too narrow. This matching problem becomes fierce and crucial when we attempt to coordinate an exponentially growing number of processors requiring an exponentially growing amount of interprocessor communication. The apparent solution to such an unmanageable problem is one that is self-organizing.

If we choose to use the graph model discussed, we are faced with a number of architecture/algorithm problems, namely, partitioning, scheduling, memory access, interprocess communication, and synchronization. The partitioning problem refers to decisions regarding the level of granularity and the choices involving which objects should be grouped into the same node of the task graph. The scheduling problem refers to the assignment of processors and memory modules to nodes of the computation graph. In general, this is an NP-complete problem (tough as nails to do optimally). The memory-access problem refers to the mechanism that allows processors to communicate with the various memory modules: usually, either shared-memory or message-passing schemes are used. The interprocessor-communication problem refers to the nature of the communication paths and connections that are available to provide processors access to the memory modules and to other processors: this may take the form of an interconnection network in a parallel-processing system, a local area network in a local distributed-processing system or shared data system or shared peripheral system, or a packet-switched, value-added, long-haul network in a nationwide distributed system. Synchronization refers to the requirement that no node in the graph model can begin execution until all of its predecessor nodes have completed their execution.

The use of broadcast or multicast communication opens up a number of interesting alternatives for communication. Local area networks take exquisite advan-

tage of these communication modes. Algorithms that require tight coupling (i.e., lots of IPC) need not only large bandwidths (which, for example, could be provided by fiber-optic channels), but also low latency. Specifically, the speed of light introduces a 15,000-microsecond latency delay for a communication that must travel from coast-to-coast across the United States.

Another consideration in matching architectures to algorithms is the balance and trade-off among communication, processing, and storage. We have all seen sys-

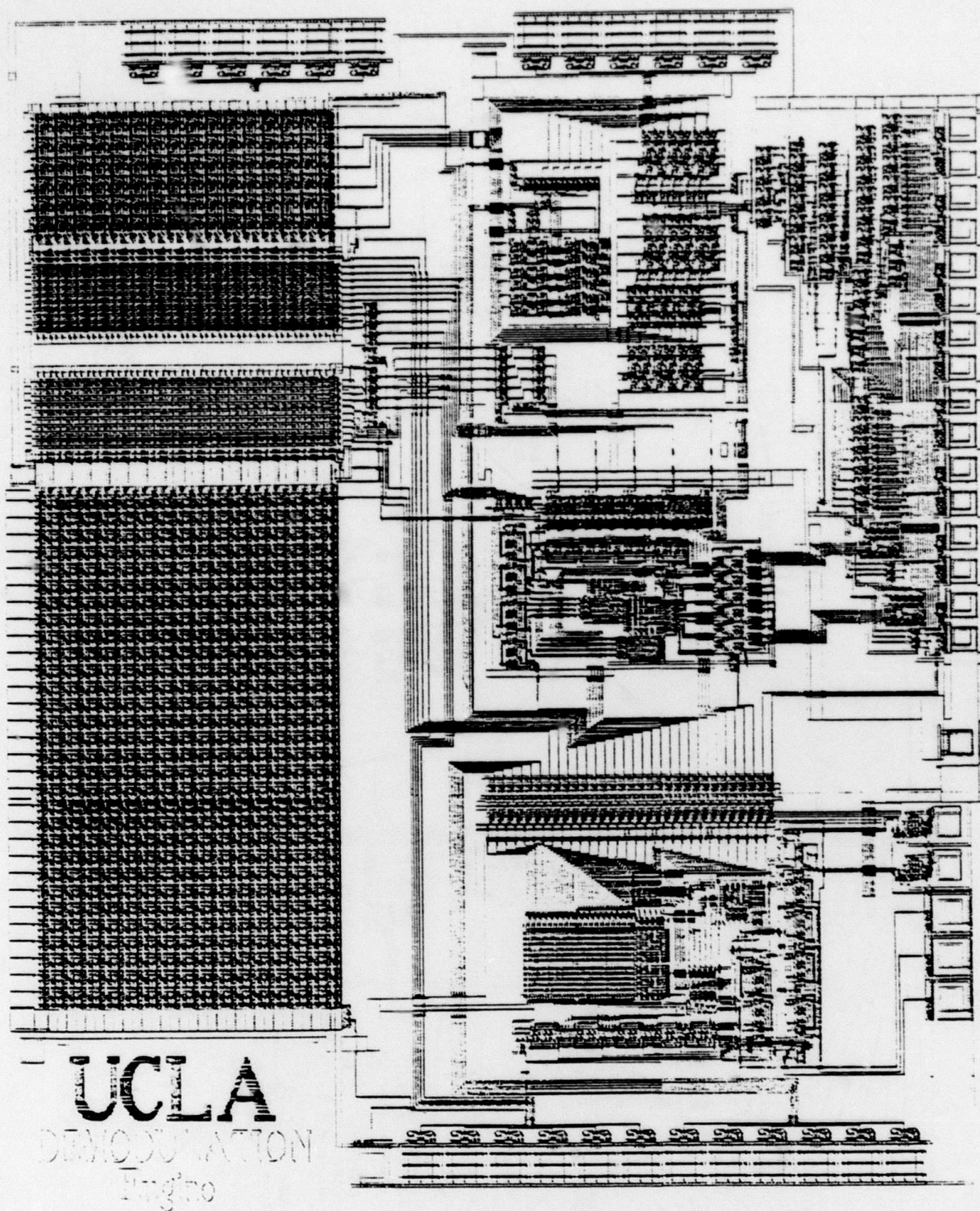


(a)

There is an amazing contrast between the neural structure of the human brain (a) and the architecture of today's VLSI chips (b). The brain is massively parallel, densely (and weirdly) connected with leaky transmission paths, highly fault tolerant, self-repairing, adaptive, noisy, and probably non-deterministic. Man-made computers are highly constrained, precisely (and often symmetrically) laid out with carefully isolated wires, not very fault tolerant, largely serial and centralized, deterministic, minimally adaptive, and hardly self-repairing. (Photo (a) is the courtesy of Peter Arnold, Inc.)

FIGURE 1. Natural and Man-Made Architectures

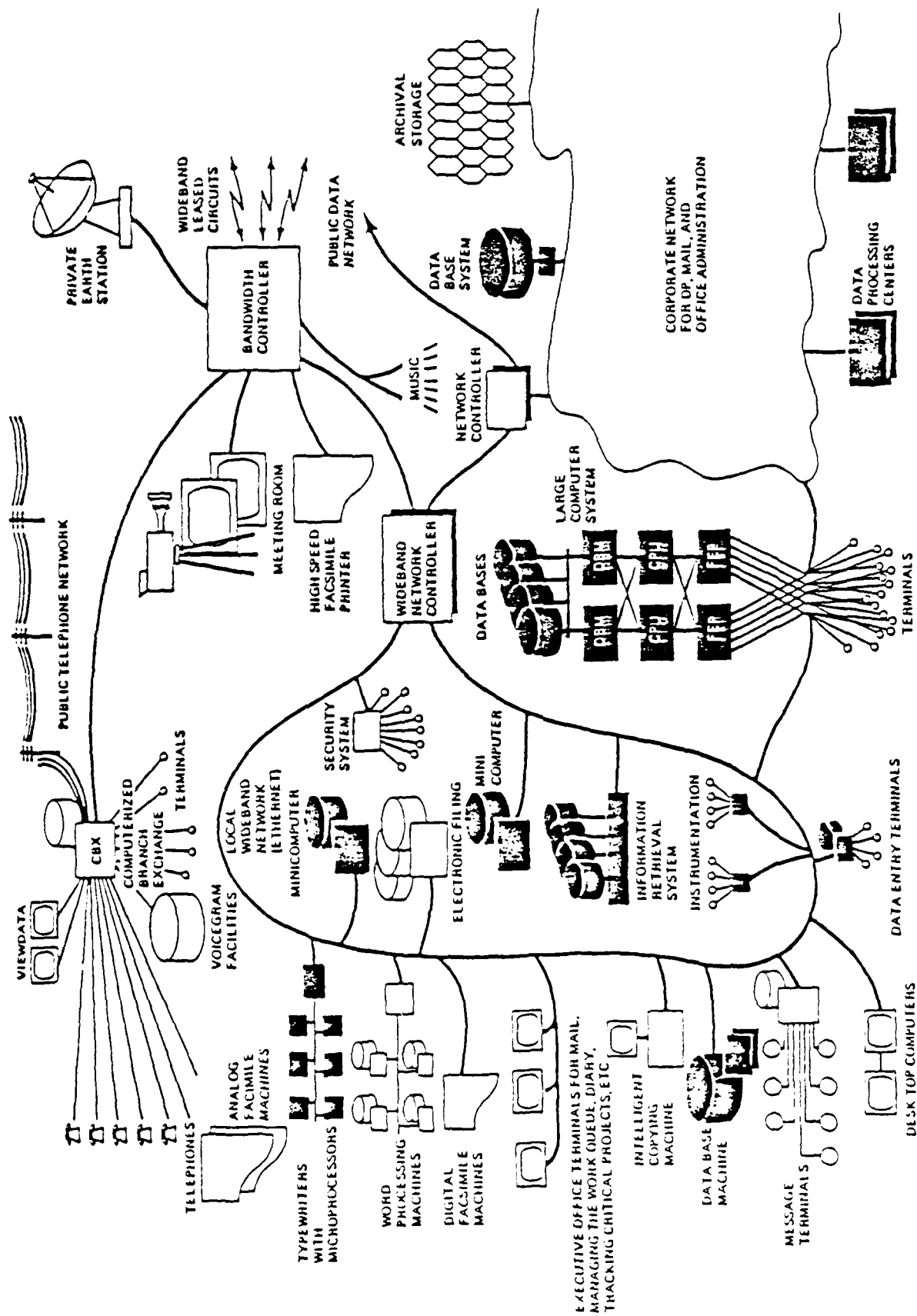
Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).



UCLA
DEMOCRATIZATION
Eng'no

(b)

FIGURE 1. Natural and Man-Made Architectures



tems where one of these resources can be exchanged for others. For example, if we do some preprocessing in the form of data compression prior to transmission, we can cut down on the communication load (trade processing for communication). If we store a list of computational results, we can cut down on the need to recompute the elements of the list each time we need the same entry (trade storage for processing). Similarly, if we store data from a previous communication, we need merely transmit the data address or name of the previous message rather than the message itself (trade storage for communication). Selecting the appropriate mix in a given problem setting is an important issue.

Distributed algorithms operating in a distributed network environment (e.g., a packet-switched network) pose the possibility that network failures may cause the network to temporarily be partitioned into two (or more) isolated subnetworks. In such a case, detection and recovery mechanisms must be introduced (see Figure 5, p. 1207).

Lastly, it should be mentioned that very little is known about characterizing those properties of an algorithm that cause it to perform well or poorly in a distributed environment.

PERFORMANCE AND BEHAVIOR

We do know some things about the way distributed systems behave, precious few though they may be. The most interesting thing about them is that they come to us from research in very different fields of study. Unfortunately, the collection of results (of which the following is a sample) is just that—a collection, with no fundamental models or theory behind it.

We begin by considering closely coupled systems, in particular, parallel-processing systems. One of the most compelling applications of parallel processing is in the area of scientific computing, where the speed of the world's largest uniprocessors is hopelessly inadequate to handle the computational complexity required for many of these problems [3]. Of course, the idea is that, as we apply more parallel processors to the computational job, the time to complete that job will drop in proportion to the number of (identical) processors, P . The "speedup" factor, denoted by S , is a common measure of performance for parallel-processing systems and is defined as the time required to complete the job using P processors, divided into the time required to complete the job using one of these processors. S may also be interpreted as the average number of busy pro-

cessors, that is, the concurrency. The best we can achieve is for S to grow directly with P ; that is,

$$S \leq P.$$

Thus, in general, $1 \leq S \leq P$. In the early days of parallel processing, Minsky [15] conjectured a depressingly pessimistic form for the typical speedup: namely,

$$S = \log P.$$

Often that kind of poor performance is indeed observed. Fortunately, however, experience has shown that things need not be that bad. For example, we can achieve $S = 0.3P$ for certain programs by carefully extracting the parallelism in Fortran DO loops [14]. However, Amdahl has pointed out a serious limitation to the practical improvements one can achieve with parallel processing (the same argument applies to the improvements available with vector machines) [1]. He argues that, if a fraction, f , of a computation must be done serially, then the fastest that S can grow with P is

$$S_{\max} = \frac{P}{fP + 1 - f}.$$

We see that, for $f = 1$ (everything must be serial), $S_{\max} = 1$; for $f = 0$ (everything in parallel), $S_{\max} = P$.

The actual amount of parallelism (i.e., S) achieved in a parallel-processing system is a quantity that we would like to be able to compute. S is a strong function of the structure of the computational graph of the jobs being processed. I, with one of my students [2], have been able to calculate S exactly as a simple function of the graph model. Specifically, we consider a parallel-processing system with P processors and with an arrival rate of λ jobs per second. We assume the collection of jobs can be modeled with an arbitrary computation graph with an average of N tasks per job, each task requiring an average of \bar{x} seconds. Then it can be shown that

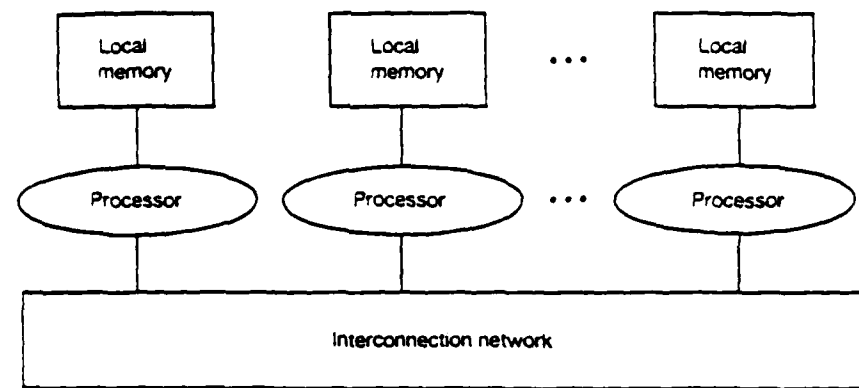
$$S = \begin{cases} \lambda N \bar{x} & \text{for } \lambda N \bar{x} \leq P \\ P & \text{for } \lambda N \bar{x} \geq P. \end{cases}$$

This is a very general result; in some special cases, the distribution of the number of busy processors can be found as well.

So far, we have given ourselves the luxury of increasing the system's computational capacity as we have added more processors to the system. Let us now consider adding more processors, but in a fashion that maintains a constant total system capacity (i.e., a constant system throughput in jobs completed per second). This will allow us to see the effect of *distributing* the computation for a job over many smaller processors. The particular structure we are considering is the regular series-parallel structure shown in Figure 6 (p. 1208), where we have taken a total processing capacity of C MIPS and divided it equally into mn processors, each of C/mn MIPS. On entering the system, a job selects

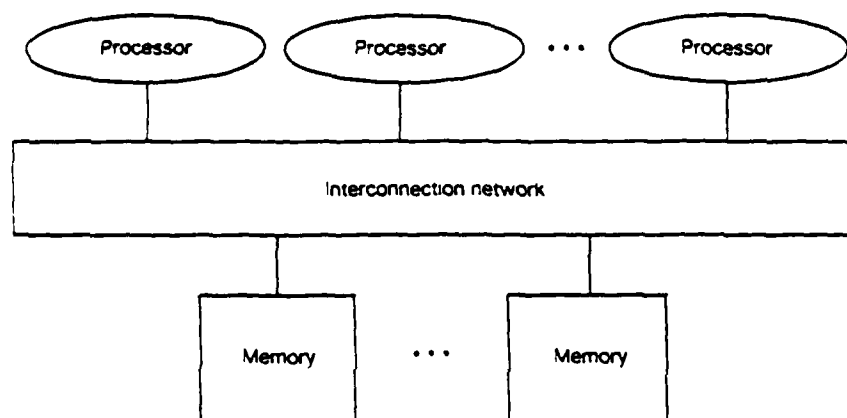
FIGURE 2. A Complex Distributed System (left)

Humans have created some unbelievably complex distributed systems. The fact that they work at all is amazing, given that we have not yet uncovered the basic principles determining their behavior. (From Martin, J. *Design and Strategy for Distributed Data Processing*. Prentice-Hall, Englewood Cliffs, N.J., 1981.)

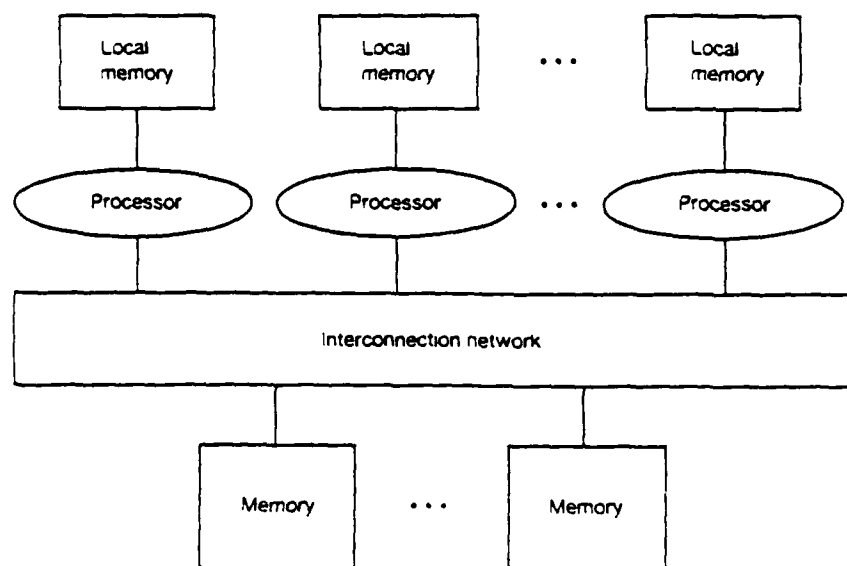


(a) Message-Passing Architecture: Local Memory

Locality of memory reference, bandwidth of communication, processor overhead, and cost are key issues determining the appropriate architecture for a given application.

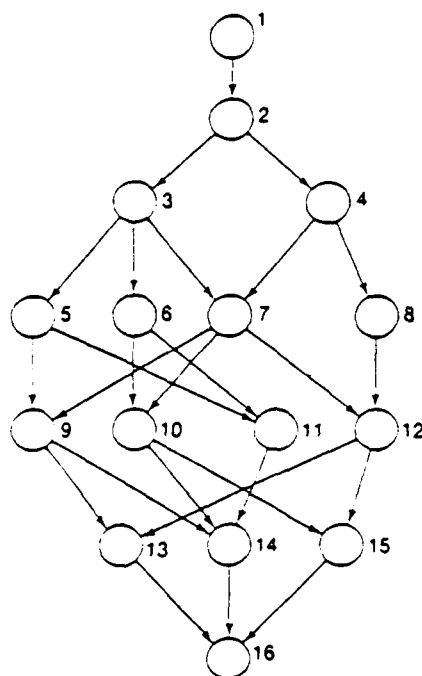


(b) Shared-Memory Architecture



(c) Hybrid Architecture

FIGURE 3. Architectures for Connecting Processors and Memory



The graph model of computation is an extremely useful model for displaying the parallelism inherent in an algorithm (i.e., a job). The entire graph represents the computational tasks associated with that job, the nodes represent the tasks themselves, and the directed arcs, which define a partial ordering of the nodes, represent the sequence in which the tasks must be performed.

FIGURE 4. Graph Model of Computation

(equally likely) any one of the m series branches down which it will travel. It will receive $1/n$ of its total processing needs at each of the n series-connected processors. The key result for this system [13] is that the mean response time for jobs in this series-parallel pipeline system is mn times as large as it would have been

had the jobs been processed by a single processor of C MIPS! There are some statistical assumptions behind this result, but the message is clear—distributed processing of this kind is terrible. Why, then, is everyone talking about the advantages of distributed processing? The answer must be that a large number of small processors (e.g., microprocessors) with an aggregate capacity of C MIPS is less expensive than a large uniprocessor of the same total capacity. It can be shown that the series-parallel system *will* have the same response time as the uniprocessor if the aggregate capacity of the series-parallel system has K times the capacity of the uniprocessor where

$$K = mn - \rho(mn - 1)$$

and where ρ is the utilization factor for each processor: namely, $\rho = \text{arrival rate of jobs times the average service time per job for a processor}$. This says that, for light loads ($\rho \ll 1$), $K = mn$, whereas, for heavy loads ($\rho \rightarrow 1$), $K = 1$. Is it the case that smaller machines are mn times less expensive than larger machines (so that we can purchase mn times the capacity at the same total price, as is needed in the light-load case)? To answer this question, recall a law that was empirically observed by Grosch more than three decades ago. Grosch's law [7] states that the capacity of a computer is related to its cost, which we denote by D (dollars) through the following equation:

$$C = \sqrt{D^2}$$

where $\sqrt{}$ is a constant. This law may be rewritten as

$$\frac{D}{C} = \frac{1}{\sqrt{C}}$$

Grosch tells us that the economics are *exactly the reverse* of what we need to break even with distributed processing! He says that larger machines are cheaper per MIPS. If Grosch is correct today, then why are microprocessors selling like hotcakes? A more recent look at the economics explains why. Ein-Dor [4] shows that, if we consider all computers at the same time, Grosch's law is clearly not true, as seen in Figure 7 (p. 1208). In this figure we see that microcomputers are a good buy.

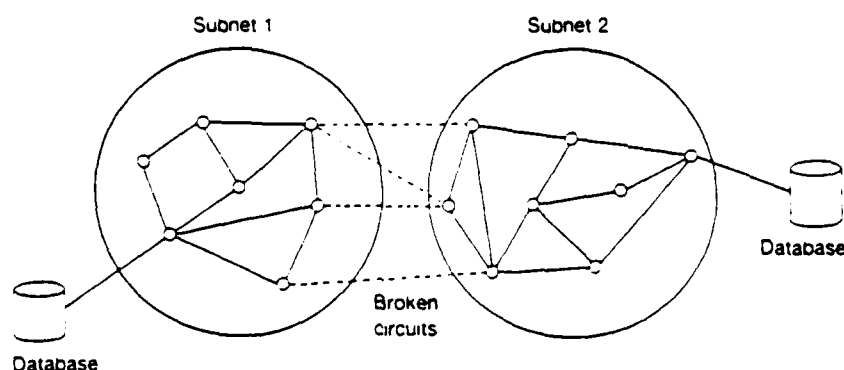
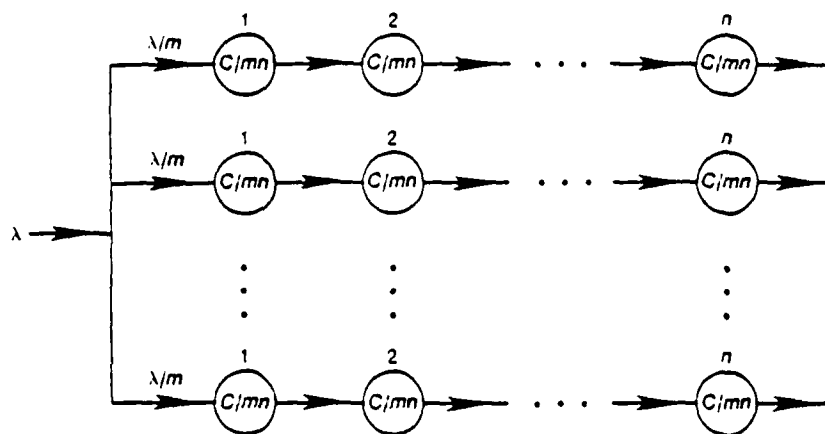


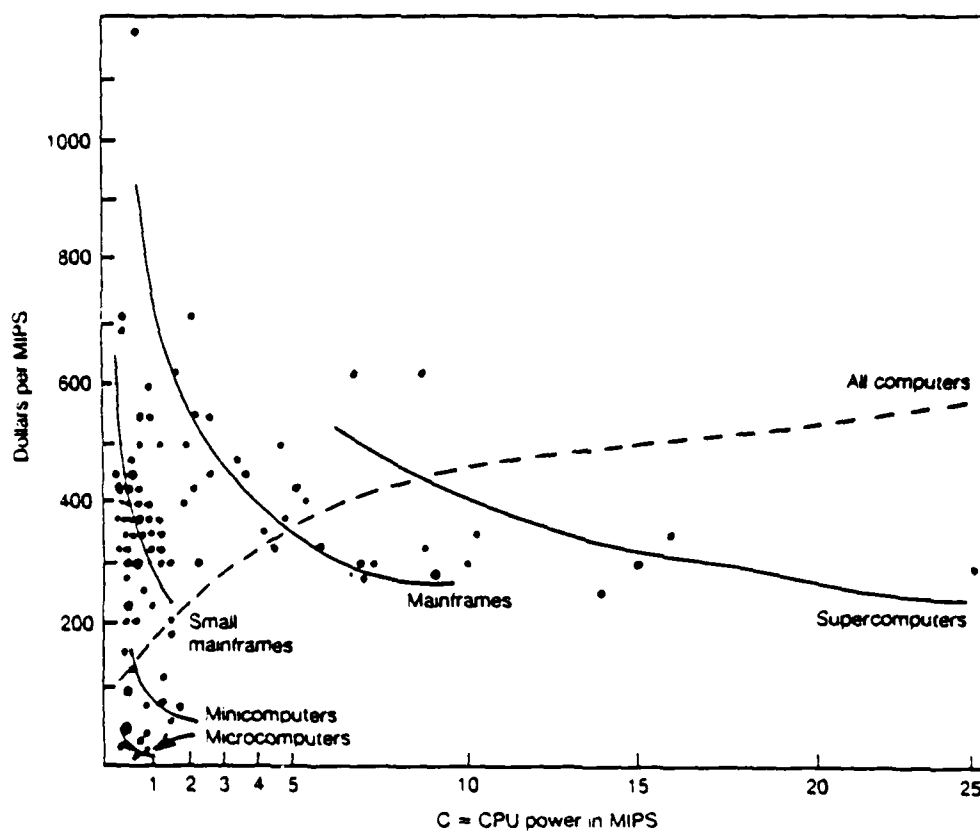
FIGURE 5. A Partitioned Network

Network failures can create two separated subnetworks that cannot communicate until the failure is repaired. Maintaining consistency of databases in such a situation is a key issue in distributed-systems design.



When a constant amount of processing capacity (C) is distributed into mn equal (and smaller) processors in a network such as this, the response time increases by a factor of mn . How can one justify a distributed system in the face of this degradation?

FIGURE 6. A Symmetrical Distributed-Processing Network



The cost per MIPS seems to rise with the number of MIPS when we examine all computers in a single group. However, when we separate them into families, we find that the opposite is true, thus confirming Grosch's law. Figure taken from

Ein-Dor, P. Grosch's law re-revisited: CPU power and the cost of computation. *Commun. ACM* 28, 2 (Feb. 1985), 142-151.

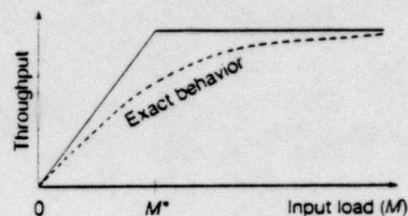
FIGURE 7. Economics of Computer Power

However, as Ein-Dor points out, Grosch's law is still true today if we consider *families* of computers. Each family has a decreasing cost per unit of capacity as capacity is increased. Ein-Dor goes on to make the observation that, if one needs a certain number of MIPS, then one should purchase computers from the smallest family that can currently supply that many MIPS. Furthermore, once in the family, it pays to purchase the biggest member machine in that family (as predicted by Grosch).

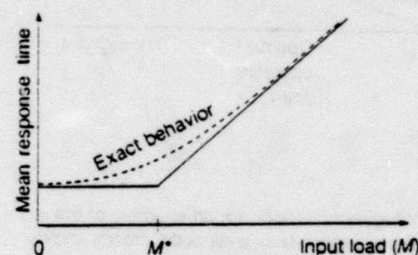
Now that we have discussed the performance of parallel-processing systems for some special cases, let us generalize the ways in which jobs pass through a multiprocessor system, and analyze the system throughput and response time. Indeed, we bound these key system-performance measures in the following way: Suppose we have a population of M customers competing for the resources of the system. Assume that customers generate jobs to be processed by some of the system's resources, that the way in which these jobs bounce around among the resources is specified in a probabilistic fashion, and that the mean response time of this system is T seconds. When a customer's job leaves the system, that customer then begins to generate another job request for the system, where the average time to generate the request is t_0 seconds. Of interest is the mean response time, T , and the system throughput γ as a function of the other system parameters. Although we have been extremely general in the system description, we can nevertheless place an excellent upper bound on the system throughput and an excellent lower bound on the mean response time as shown in Figure 8. In this figure, the quantity M^* is defined as the ratio of the mean cycle time $T_0 + t_0$ to the mean time x_0 required on the critical resource in a cycle; T_0 is the mean response time when $M = 1$, and the critical resource is that system resource that is most heavily loaded [11].

To find the exact behavior (shown in dashed lines in the figure) rather than the bounds, one must be much more explicit about the distributions of the service time required by jobs at each resource in the system as well as the queueing discipline at each. Using the bounds or the exact results, the effect of parameter changes on the system behavior can be seen. For example, one can examine the accuracy of the common rule of thumb that suggests that the proper mix of microprocessor speed, memory size, and communication bandwidth is in the proportion 1 MIPS, 1 Mbyte, and 1 Mbit per second; some suggest that we will soon see a 10, 10, 10 mix instead of the 1, 1, 1 mix. Of course the correct answer to this question depends on the total system configuration.

Once we evaluate the throughput and mean response time for a system, we usually want to find the relationship between the two, which typically has the well-known shape (shown in Figure 9, p. 1210) that clearly demonstrates the trade-off between them—a low delay implies a small throughput and vice versa.



(a) Bound on Throughput



(b) Bound on Mean Response Time

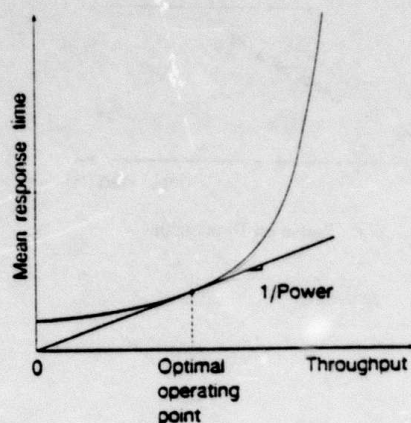
Excellent bounds on throughput (a) and mean response time (b) as a function of the number of users (or any measure of the input load) are easily obtained for a very large class of distributed systems. The exact behavior can be derived for more restricted systems and demonstrates the excellence of the bounds.

FIGURE 8. Bounds on Throughput and Response Time

We are immediately compelled to inquire about the location of the "optimal" operating point for a system. The answer depends on how much you hate delay versus how much you love throughput. One way to quantify this love-hate choice is to define a quantity known as "power" (denoted by P), which is defined as

$$P = \frac{\gamma}{T}$$

The operating point that optimizes (i.e., maximizes) the power (large throughput and small delay) is located at that throughput where a straight line (of minimum slope) out of the origin touches the throughput-delay profile (usually tangentially); such a tangent and operating point are shown in Figure 9. This result holds for all profiles and all flow-control functions (see below). Moreover, for a large class of queueing curves, this optimal operating point implies that the system should be loaded in such a way that each resource has, on the average, exactly one job to work on [12].



The delay-throughput relationship, an example of the key profile in systems performance evaluation, clearly shows the trade-off between the two. In general, you cannot get a small delay and a large throughput at the same time. We can, however, maximize "power," which is the ratio of throughput to delay, in order to define the natural point for a system.

FIGURE 9. The Key System Profile

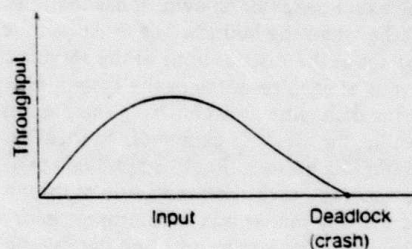
Unfortunately, there are some distributed systems that do not have the nice relationship shown in Figure 8a where the throughput rises asymptotically to its maximum value as the "input" is increased. Often we find the behavior depicted in Figure 10 where the throughput reaches a peak and then declines as the input increases further, possibly dropping to zero, in which case we say that the system has crashed. Such behavior has been observed in paged virtual-memory systems (thrashing), in computer networks (deadlocks and degradations), and in automobile traffic flow (bumper-to-bumper traffic). Here again, one must find a method for controlling the input (i.e., setting the system operating point) so as to achieve optimal or near-optimal performance (somewhere near the peak of the curve in Figure 10).

"Flow control" is the name associated with this operation, and it can be implemented in a centralized or a distributed fashion in distributed systems with the latter being the more challenging design problem [6]. One example of distributed control is the dynamic routing procedure found in many of today's packet-switching networks where no single switching node is responsible for the network routing. Instead, all nodes participate in the selection of network routes in a distributed fashion. A great deal of research is currently under way to evaluate the performance of other distributed algorithms in

networks and distributed systems. Examples are the distributed election of a leader, distributed rules for traversing all the links of a network, and distributed rules for controlling access to a database.

Another large class of distributed-control algorithms has to do with sharing a common communication channel among a number of devices in a distributed fashion [19]. If the channel is a broadcast channel (also known as a one-hop channel), then the analytic and design problem is fairly manageable and a number of popular local area network algorithms for media access control have been studied and implemented. Examples here include CSMA/CD (carrier-sense multiple access with collision detect—as used in Xerox's Ethernet, AT&T's 3B-Net and Starlan, and IBM's PC Network), token passing (as used in the token-ring and token-bus networks), and address contention resolution (as used in AT&T's ISN). A large number of additional channel access algorithms have been studied in the literature including Expressnet, tree algorithms, urn models, and hybrid models. If the channel is multicast (or multi-hop), then the analytic problem becomes much harder.

But what if the processors in our distributed environment are allowed to communicate with their peers in very limited ways? Can we endow these processors (let us call them automata for this discussion) with an internal algorithm that will allow them to achieve a collective goal? Tsetlin [20] studied this problem at length and was able to demonstrate some remarkable behavior. For example, he describes the Goore game in which the automata possess finite memory and act in a probabilistic fashion based on their current state and the current input. They cannot communicate with each other at all and are required to vote YES or NO at



There are many systems that degrade badly when pushed too hard. They can even degrade to a situation of deadlock. Examples include thrashing in virtual memory systems, deadlocks in computer networks, and bumper-to-bumper traffic in highway systems.

FIGURE 10. A Dangerous Throughput Profile

certain times. The automata are not aware of each other's vote; however, there is a referee who can observe and calculate the percentage, p , of automata that vote YES. The referee has a function, $f(p)$ (such as that shown in Figure 11), where we require that $0 \leq f(p) \leq 1$. Whenever the referee observes a percentage, p , who vote YES, he or she will, with probability, $f(p)$, reward each automaton, independently, with a one dollar payment; with probability $1 - f(p)$ he or she will punish an automaton by taking one dollar away. Tsetlin proved that no matter how many players there may be in a Goore game, if the automata have sufficient memory, then for the payoff probability shown in the figure, exactly 20 percent of the automata will vote YES with probability one! This is a beautiful demonstration of the ability of a distributed-processing system to act in an optimum fashion, even when the rules of the reward function are unknown to the players and when they can neither observe nor communicate with each other. All they are allowed is to vote when asked, and to observe the reward or penalty they receive as a result of that vote. In this work we see the beginnings of a theory that may be able to explain how the colony of ants performs its tasks.

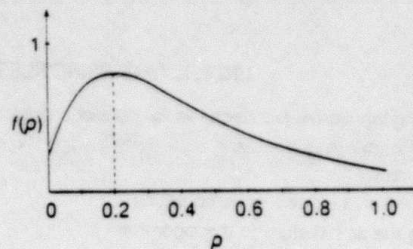
NEEDED UNDERSTANDING AND TOOLS

In the previous section, we discussed a few of the things known about distributed-systems performance and behavior. A few isolated facts are indeed known, but overall theory and understanding are still lacking.

For instance we need considerably sharper tools to evaluate the ways in which randomness, noise, and inaccurate measurements affect the performance of distributed systems. What is the effect of distributed control in an environment where that control is delayed, based on estimates, and not necessarily consistent throughout the system? What is the effect on performance of scaling some of the system parameters? We need a common metric for discussing the various system resources of communications, storage, and processing. For example, is there a processing component to communications? We also need a proper way to discuss distributed algorithms and distributed architectures.

A microscopic theory that deals with the interaction of each job with each component of the system is likely to overwhelm us with detail and will fail to lead us to an understanding of the overall system behavior. It is similar to the futility of studying the many-body problem in physics in order to obtain the global behavior of solids. What is needed is a macroscopic theory of distributed systems, such as thermodynamics has provided for the physicist. In fact, Yemini [21] has proposed an approach for a macroscopic theory based on statistical mechanics that will lead to better understanding the global behavior of distributed systems without the need for a detailed, fine-grained analysis.

Another fruitful approach that also avoids the horrible details of any specific system structure must be credited to Shannon [18]. In analyzing the behavior of



The Goore game rewards each member of a set of automata independently with a probability given by the function $f(p)$, where p is the fraction of the set that votes YES at a given time. The automata are completely unaware of the other automata, do not know the function $f(p)$, and, remarkably, will collectively vote in a way that maximizes the payoff to all.

FIGURE 11. The Goore Game

error-correcting codes for noisy communication channels. Shannon used the brilliant device of studying *all possible codes simultaneously*. This enabled him to average out the detailed structure of any given code. He could then take exquisite advantage of the law of large numbers in order to arrive at a precise statement regarding the error behavior of codes. It is likely that such an approach will allow us to study the behavior of "typical" topologies and algorithms in distributed systems.

LIKELY FUTURE DEVELOPMENTS

These are exciting times. Researchers in universities and laboratories around the world have begun to focus their attention on distributed systems. They come to this field from diverse disciplines ranging from queueing theory to neuroanatomy in which they are the experts. Thus, we have the ingredients for an enormously rich soup of separate ideas that have only just begun to blend.

As the theoretical frontiers are being assaulted, so too are the practitioners busily building systems. This is a double-edged sword. On the one hand, the implementation of real distributed systems in the hands of the designers and users provides us with a strong motivation for progress in understanding, as well as a magnificent test bed in which we can experiment. On the other hand, these systems are massively expensive and are being implemented without the benefit of the principles we seek. As a result, they may be colossal failures! The reality is that there is no way we can prevent their proliferation as manufacturers respond to the frenzied demand from the user community. In a sense

UNDERLYING PRINCIPLES OF DISTRIBUTED-SYSTEMS BEHAVIOR

- Developing innovative architectures for parallel processing
- Providing better languages and algorithms for specification of concurrency
- More expressive models of computation
- Matching the architecture to the algorithm
- Understanding the trade-off among communication, processing, and storage
- Evaluation of the speedup factor for classes of algorithms and architectures
- Evaluation of the cost-effectiveness of distributed-processing networks
- Study of distributed algorithms in networks
- Investigation of how loosely coupled self-organizing automata can demonstrate expedient behavior
- Development of a macroscopic theory of distributed systems
- Understanding how to average over algorithms, architectures, and topologies to provide meaningful measures of system performance

we are all responsible for the current craziness, because we have been "promising" these miraculous systems to the user for almost a decade.

In the face of these developments, we can foresee some of the likely developments that will take place over the next decade or two. Let us first consider the likely technology developments in hardware-type resources. One of the most exciting of these is the huge data bandwidth projected for fiber-optic technology. These fibers are being used for point-to-point communication pipes at rates on the order of hundreds of megabits per second. Bell Laboratories and Japan have been leapfrogging each other in setting world records for the largest data rates transmitted over the longest distances. Earlier this year, Bell Laboratories established a new record by transmitting at the rate of 4 billion bits per second at a distance of 117 km without any repeaters! The product of data rate times distance has been doubling every year since 1975, and based on the limits imposed by physics, there are still five orders of magnitude to go (16 years of doubling left). The tiny glass fiber is so clear that, if the oceans of the world were made of this glass, one could see the bottom of the deepest trench in the ocean floor from the surface. If we consider a 1-mW laser and a requirement of 10 photons to detect 1 bit of information (high-quality detection), then a single strand of fiber should be able to support a data bandwidth of 10^{15} bits per second. That would provide, for example, a 100-Mbit-per-second channel to each of 10 million users—all on one thin strand! This light-wave technology is being installed across the United States right now. The Los Angeles 1984 Olympics video was transmitted from the games' remote locations to satellite transmitters using a fiber-optic network installed by Pacific Bell—perhaps the most well-known application to date. This technology is being applied to local area networks by a number of vendors, but the technology for this application is not yet mature because we have yet to develop an efficient way to optically tap into the light pipes at low loss. As soon as that problem is resolved (in the next two or three years), we are likely to see a rapid deployment of fiber-optic channels in our local network environment.

As discussed earlier, enormous bandwidths are necessary, but not sufficient, for many tightly coupled systems. The latency introduced due to propagation delay can inhibit tight control. (E.g., if we transmit data into a 1-Gbit-per-second light pipe spanning the United States, the 15,000-microsecond propagation delay is such that the first bit will come out of the other end only after 15 million bits have been pumped in!)

This planet is currently laced with many types of computer/communications networks at all levels. There are wide area networks, packet-switched networks, circuit-switched networks, satellite networks, packet radio networks, metropolitan area networks, local area networks, cellular radio networks, and more; and they are mostly incompatible within each type and across types. At the same time, the end user's facility consists of telephones, data terminals, host machines, PBX switches, alarm systems, video systems, FAX machines, etc. The incompatibility problem escalates! What is needed in a distributed system is a standard digital communication service to connect the many user devices with one another across the room or across the world. Fortunately, there is a worldwide movement to define and adopt an integrated solution to this problem, which has given rise to the Integrated Services Digital Network (ISDN). The ISDN service defines a customer interface (a plug in the wall) to which the user's devices can attach and gain access to the worldwide integrated digital network. We are not likely to see much definition and penetration of ISDN until the end of this decade and, possibly, into the next decade (and most likely it will first appear at the local network level).

What all this should tell us is that we are approaching a time when massive connectivity among devices and systems will exist. Such connectivity is necessary if we are to derive the full benefits from distributed systems.

At the processor technology level, perhaps the most dramatic development is the gathering momentum in the proliferation of personal workstations. They are spearheading the drive toward distributed systems. At the other end of the spectrum, parallel machine archi-

tures are being proposed all over the world to increase the processing capacity that can be applied to a single problem. Both of these technologies are moving very rapidly and are putting pressure on distributed-systems research and development. We are seeing the development of massively distributed architectures that can be configured as tightly coupled, loosely coupled, or even hierarchically structured systems.

Massively distributed and massively connected systems with enormous computational capacity are likely to appear in the next 10 years. Unless we pay very careful attention to the user interface, users will be hopelessly lost and ineffective. At the very least, we must provide users with languages that allow them to take advantage of the distributed architecture and to write application code quickly and in a way that allows the application package to be modified and maintained easily. Moreover, the complexity of the system should be transparent to users. Users need to interface with a systemwide operating system that offers the use of a single logon (with a networkwide name and password) and that provides access to file servers, database servers, automatic backup, processing servers, mail servers, application packages, education and help functions, etc.

The system itself could take advantage of expert-systems capability in providing these services to the user. And the system is likely to include extensive redundancy in order to provide high levels of reliability and fault tolerance. It should also be self-repairing, and even self-organizing, as the conditions and demands on it change.

Aside from the business-oriented applications and developments listed above, an enormous consumer-oriented set of products will be developed. One device that spans business and personal needs is a proper "lap" computer that will provide the user with remote access to the massive distributed network resources described in this article.

We foresee a new phenomenon whereby users are confronted with so many attractive features in new devices and software packages that they cannot possibly learn to use them all. Learning how to use the features represents an investment far beyond users' available time; and yet the features are wonderfully seductive. To coin a term, I would like to refer to this phenomenon as "FEATURE SHOCK".

As we observe the growth of our man-made distributed systems, we wonder how the ants, bees, birds, fish, and higher animals have managed to perform so well with their distributed systems. If we are ever to achieve a level of performance anywhere near theirs, we will have to further uncover the underlying principles of distributed-systems behavior (see sidebar). We have discussed some of these in this article, but there is much new ground to be broken. Almost anywhere you dig you are likely to find pay dirt. The field is wide open for new ideas and new approaches, challenging problems remain unsolved, and the application of new results will be widespread and rapid—what lovelier environment could you seek?

REFERENCES

1. Amdahl, G.M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS*, Vol. 30, Thompson, Washington, D.C., 1967, pp. 483-485.
2. Beighith, A., and Kleinrock, L. Analysis of the number of occupied processors in a multi-processing system. UCLA CSD Rep. 850027, Computer Science Dept., Univ. of California, Los Angeles, Aug. 1985.
3. Denning, P.J. Parallel computation. *Am. Sci.* (July-Aug. 1985).
4. Ein-Dor, P. Grosch's law re-revisited: CPU power and the cost of computation. *Commun. ACM* 28, 2 (Feb. 1985), 142-151.
5. Ercegovac, M., and Lang, T. General approaches for achieving high speed computations. In *Supercomputers*, S. Fernbach, Ed. North-Holland, Amsterdam. To be published.
6. Gerla, M., and Kleinrock, L. Flow control protocols. In *Computer Network Architectures*, Paul Green, Ed. Plenum, New York, 1982, pp. 361-412.
7. Grosch, H.A. High speed arithmetic: The digital computer as a research tool. *J. Opt. Soc. Am.* 43, 4 (Apr. 1953).
8. Grossberg, S. *Studies of the Mind and Brain: Neural Principles of Learning, Perception, Development Cognition and Motor Control*. Reidel, Hingham, Mass., 1982.
9. Hwang, K. Multiprocessor supercomputers for scientific/engineering applications. (June 1985), 57-73.
10. Hwang, K., and Briggs, F. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.
11. Kleinrock, L. *Queueing Systems, Volume 2: Computer Applications*, Chap. 4. Wiley-Interscience, New York, 1976.
12. Kleinrock, L. On flow control in computer networks. In *IEEE Proceedings of the Conference in Communication*, Vol. 2, IEEE, New York, June 1978, pp. 27.2.1-27.2.5.
13. Kleinrock, L. On the theory of distributed processing. In *Proceedings of the 22nd Annual Allerton Conference on Communication, Control and Computing*, Univ. of Illinois, Monticello, Oct. 1984, pp. 60-70.
14. Kuck, D.J. et al. The effects of program restructuring, algorithm change and architecture choice on program. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1984, pp. 129-138.
15. Minsky, M., and Papert, S. On some associative, parallel and analog computations. In *Associative Information Technologies*, E.J. Jacks, Ed. Elsevier North Holland, New York, 1971.
16. Patton, C.P. Microprocessors: Architecture and applications. *IEEE Comput. Mag.* 18, 6 (June 1985), 29-40.
17. Schneek et al. Parallel processor programs in the federal government. *IEEE Comput. Mag.* 18, 6 (June 1985), 43-56.
18. Shannon, C., and Weaver, W. *The Mathematical Theory of Communication*. Univ. of Illinois Press, Urbana, 1962.
19. Stuck, B.W., and Arthurs, E. *A Computer and Communications Network Performance Analysis Primer*. Prentice-Hall, Englewood Cliffs, N.J., 1985.
20. Tsetlin, M.L. *Automaton Theory and Modeling of Biological Systems*. Academic Press, New York, 1973.
21. Yemini, Y. A statistical mechanics of distributed resource sharing mechanisms. In *Proceedings of INFOCOM 83*, 1983, pp. 531-539.

CR Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems

General Terms: Design, Performance, Theory

Additional Key Words and Phrases: computer networks, distributed control, distributed processing, economics of computing power, machine architecture, multiprocessing, parallel processing, performance of distributed systems, self-organizing systems

Author's Present Address: Leonard Kleinrock, Dept. of Computer Science, Boelter Hall, UCLA, Los Angeles, CA 90024.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

BROADCAST COMMUNICATIONS AND DISTRIBUTED ALGORITHMS

Rina Dechter

Leonard Kleinrock

Abstract

The paper addresses ways in which one can use "broadcast communication" in distributed algorithms and the relevant issues of design and complexity. We present an algorithm for merging k sorted lists of $\frac{n}{k}$ elements using k processors and prove its worst case complexity to be $2n$, regardless of the number of processors, while neglecting the cost arising from possible conflicts on the broadcast channel. We also show that this algorithm is optimal under single-channel broadcast communication. In a variation of the algorithm we show that by using an extra local memory of $O(k)$ the number of broadcasts is reduced to n . When the algorithm is used for sorting n elements with k processors, where each processor sorts its own list first and then merging, it has a complexity of $O(\frac{n}{k} \log \frac{n}{k} + n)$, and is thus asymptotically optimal for large n . We also discuss the cost incurred by the channel access scheme and prove that resolving conflicts whenever k processors are involved introduces a cost factor of at least $\log k$.

1. Introduction

Consider the following algorithm for finding the maximum of a set of k distinct numerically valued elements where each element is stored within a separate processor. When the algorithm begins, each processor attempts to transmit its own value using a common broadcast channel to which all processors listen. However, only one processor is enabled (permitted) to transmit by means of some access scheme (conflict resolution scheme). Each processor compares its value with the largest value transmitted so far. All processors that have a larger value try again to

broadcast their own values, etc. The algorithm terminates when all processors have either transmitted their values or have "given up" which is detected by silence on the channel. The last element to be broadcast is the maximum.

This admittedly simple algorithm (referred to as the "Max-Algorithm" and also presented in [16]) demonstrates how a distributed algorithm can utilize broadcast communication. The term broadcast implies the existence of a single channel on which only one node (processor) can transmit at one time while all the others receive the message simultaneously.

"Algorithms by broadcasting" have not received much attention in the literature on parallel and distributed algorithms. An earlier report by the authors [8] was among the first discussions of such algorithms and those results constitute a portion of the current paper. Other contributions, appearing at around the same time, are [15, 16]. In this introduction we survey the motivation for using broadcasting as a model for distributed computation, point out its unique features, summarize relevant work and point out our contribution.

In the area of parallel algorithms, the closest thing to broadcasting is the assumption of the existence of a global, or a shared, memory from which all the processors can simultaneously read the same value [5, 20]. However, shared memory models usually do not place any limit on the number of memory cells which are used by the processors. In the context of broadcasting this would mean that there is more than one broadcast channel and that each processor can use any channel according to the requirements of the algorithm. In most broadcast-based networks (e.g. local area networks [18]) there is only one channel shared by all processors. Therefore, most of the results for shared memory models are not applicable to broadcasting. An example in which the results are applicable is the search algorithm presented by Snir [20] using the CREW (concurrent read, exclusive write) model which utilizes only one memory location.

A major difficulty in using broadcast communication is the issue of access to the channel. Many access schemes have been proposed and analyzed [22]. The focus of most papers is on how to increase channel capacity and on the tradeoff between throughput and delay. Clearly, the access scheme may have a significant impact on the complexity of the algorithm. We will approach this problem through two models. In both models, processors broadcast one value at a time on a channel shared by all of them. Only one message will be posted at a slot on the channel, and all processors can read the posted message. In our first model, named IPABM (Ideal Parallel Broadcast Model), we assume that some "ideal" access scheme exists, i.e., if several processors demand the use of the channel at the same time, there is a global mechanism which enables one of them to transmit in a constant time. Later it is refined into a "more realistic" model, RPABM (Realistic Parallel Broadcast Model), that incorporates a conflict resolution protocol (CRP), and we discuss two specific access schemes and their influence on the time complexity of the algorithms.

The vehicle we use to study algorithms by broadcasting is via "comparison based" algorithms (sorting, searching, etc.). Parallel versions of these algorithms have been extensively studied under various models of communications [2, 3, 5, 7, 19, 20, 21, 23, 24] and therefore they are well suited for studying the power and limitations of broadcast communication. For a review of sorting algorithms see also [9].

Relevant work in the area of broadcast algorithms includes the work by Levitan [15] who uses a PBM (Broadcast Protocol Multiprocessor) model which is identical to our IPABM, and obtains results similar to those given in the current paper: in particular he presents a sorting algorithm which is identical to our second version of the merge algorithm when all processors have just one element. In addition, he gives an algorithm for finding a minimum spanning tree in a graph. Algorithms for finding the extrema in a broadcast model are presented in [16] and [4]. The latter uses a mixed model that, in addition to the conventional links, also allows a global bus for

broadcast communication.

The main contributions of this paper are: an efficient algorithm for merging, proving its optimality and dealing in a formal way with issues that emerge from the use of broadcasting as a model for distributed computing. In particular, in the analysis we take into account both the communication cost (time to broadcast a message) and the computation cost (time for performing a comparison). We also discuss the overhead introduced by different access schemes.

In the next section we present our model, discuss complexity issues and analyze the performance of the Max-algorithm discussed above. In Section 3 we present an algorithm for merging k sorted lists of $\frac{n}{k}$ elements each. We show that the worst case performance of the algorithm is independent of the number of processors (which is also the number of lists), and is bounded from above by $2n-1$ broadcasts and the same number of comparison stages. A comparison stage is one time slot in which several processors in parallel perform one comparison. In a variation of this algorithm we show (Section 3.3) that an additional $O(k)$ storage in each processor can reduce the number of messages broadcasted to n . Using the merge algorithm for sorting n elements with k processors (Section 3.4) yields a worst case time complexity of $O(\frac{n}{k} \log \frac{n}{k} + n)$. Thus for large n , the Merge-sort algorithm achieves an asymptotic speed-up ratio of k with respect to the best sequential (i.e. single processor) algorithm whose complexity is $O(n \cdot \log n)$. In Section 4 we show the optimality of the Merge algorithm by proving that any Merge-by-broadcast algorithm requires n broadcasts. Section 5 addresses access scheme issues.

2. The IPABM model and complexity measures

The model which is used through the most of the paper is presented next. Let us define an IPABM as a collection of processors which compute in parallel synchronously and which communicate via a single broadcast channel. The channel is slotted into time slots of size T (where T is the time for a message transmission). At each step each processor can read the message in the

current slot on the channel, do some computation and submit a message to be broadcast in the next time slot. Any number of processors can read the current message on the channel but only one message among those submitted for transmission will be chosen by the global access mechanism to be broadcast in the next time slot. An empty slot indicates that no processor wants to talk.

The complexity of an algorithm will be measured by its computing time and its communication time. Dealing with comparison based algorithms, we consider a comparison operation as the basic computation step and the broadcast of a message as the basic communication step. Thus the "number of comparisons" (#comparisons) performed in parallel and the "number of broadcasts" (#broadcasts) characterize the computation time and broadcast time, respectively. Let t be the time for a comparison operation (since the algorithms we consider are synchronized the analysis does not take into account variations in computing time among processors). The above two measures are combined as follows:

$$T(A) = t \cdot (\#comparisons) + T \cdot (\#broadcasts) \quad (1)$$

where $T(A)$ stands for the worst case time complexity of algorithm A. By $\overline{T(A)}$ we denote the average time complexity of algorithm A over all problem instances.

In the Max-Algorithm each broadcast is followed by one comparison operation performed in parallel by some of the processors. Therefore it is sufficient to account only for #broadcasts as the measure of complexity.

Let $T(\text{Max}(k))$ be the number of broadcasts performed by the Max algorithm with k elements and k processors. On some input instances $\text{Max}(k)$ will require each element to be broadcast and thus

$$T(\text{Max}(k)) = k. \quad (2)$$

The average number of broadcasts, $\overline{T(Max(k))}$, obeys the following recurrence:

$$\overline{T(Max(k))} = 1 + \frac{1}{k} \sum_{i=1}^k \overline{T(Max(k-i))} \quad (3)$$

This last is true since if the first element to be broadcast is the i^{th} smallest element, then exactly $i-1$ among the rest of the $k-1$ processors will remain silent and will not participate in the rest of the algorithm. We assume a homogeneous distribution of the elements among the processors and thus the above event has probability $\frac{1}{k}$ and the recurrence follows. (3) can be written as:

$$\overline{T(Max(k))} = 1 + \frac{1}{k} \sum_{i=0}^{k-1} \overline{T(Max(i))} \quad (4)$$

with $\overline{T(Max(0))} = 0, \overline{T(Max(1))} = 0$. Solving this recurrence yields

$$\overline{T(Max(k))} = \sum_{i=1}^{k-1} \frac{1}{k} \leq \log k \quad (5)$$

The same results were obtained in [16] using a slightly different analysis.

3. Parallel Merge by Broadcast

3.1 Description of the algorithm

We present a distributed algorithm that merges k sorted lists of $\frac{n}{k}$ distinct elements into a decreasing series, using an IPABM with $k-1$ processors. Each of the first k processors contains one of the lists and each has an identity (id#) and a local memory. The size of local memory is fixed and not dependent on k or n . For simplicity we designate the $(k+1)^{th}$ processor to be the "output processor" (the one in which the output will be stored.) All processors cooperatively participate in the task of merging the sorted lists they possess. The maximum element in each processor's list is called its "current value".

The algorithm can be decomposed into cycles. In each cycle the maximum of the current values is determined. This element is broadcast to the output processor as the next element in the merged list, and is removed from the processor to which it belonged (the processor updates its

current value). Each cycle is implemented by the Max-Algorithm presented earlier. The processor that broadcasts first is the initiator of the cycle; the last, is the terminator of the cycle. During the cycle processors try to broadcast their current values as long as they haven't heard yet a larger value being broadcast. In order to eliminate redundant broadcasts there will be some dependency between the initiations of cycles.

When a processor succeeds in broadcasting its current value, it denotes the value which is broadcasted immediately afterwards as its successor. The successor value is updated each time the current value is rebroadcast. When the current value is the terminator of the cycle, it has no successors. The current value is the predecessor of its successor. Each current value will have at most one successor at any given time (it may have none, if it was not broadcast yet). It will also have at most one predecessor. In terms of this terminology, the rule for cycle initiation is as follows: a processor initiates the next cycle (by rebroadcasting its current value) if the present cycle was terminated by a successor to its own current value. If there is no predecessor the next cycle can be initiated by any processor.

In order to implement the above algorithm in a distributed fashion, processors must be able to detect the end of a cycle and its terminator, and to determine whether or not they should initiate the next cycle. The end of a cycle is determined by silence (i.e. an empty time slot). The initiator of a cycle is determined by the successor-predecessor relationship, as described earlier. Two empty slots in succession indicate that a cycle is terminated but that a specific initiator does not exist, in which case all processors try to initiate the next cycle.

The algorithm for each processor is described in Figure 1. While listening to the channel a processor can recognize one of the following three cases: A value is broadcast in the current slot, (case 1) or, the current slot is empty but the previous one holds a value (case 2), or two consecutive empty slots have occurred. In cases 2 and 3 a cycle has terminated and an initiator must be determined. We assume that each processor has a procedure for determining the terminator of a

cycle which is used in case 2. The procedure `process_update` is used each time the processor has successfully broadcasted its current value, CV. It determines whether the value is also the terminator of the cycle, or whether, the successor value, SUCC, should be updated. If it is the terminator, the value at the top of its list is removed and the value of CV is updated to be the new maximum in its list. The first broadcast of each current value may terminate the cycle in which it participated. If it didn't, this value will be rebroadcast only for initiating future cycles until it will terminate one. Then, the current value is updated and the processor tries to broadcast the new current value for the first time. A formal proof of this behavior is given later.

It is convenient to trace the execution of the algorithm using a global work stack in which the values being broadcast are recorded. The work stack could also be kept in each of the processors and thus be used to control the algorithm (see Section 3.3). Here it is utilized only to explain the rule for cycle initiation.

Consider the following example. Suppose we have $n=8$, $k=4$ and the initial situation is as follows:

| | | | |
|--|--|--|--|
| $\begin{array}{ c } \hline 63 \\ \hline \end{array}$ | $\begin{array}{ c } \hline 79 \\ \hline \end{array}$ | $\begin{array}{ c } \hline 84 \\ \hline \end{array}$ | $\begin{array}{ c } \hline 66 \\ \hline \end{array}$ |
| $\begin{array}{ c } \hline 54 \\ \hline \end{array}$ | $\begin{array}{ c } \hline 64 \\ \hline \end{array}$ | $\begin{array}{ c } \hline 75 \\ \hline \end{array}$ | $\begin{array}{ c } \hline 65 \\ \hline \end{array}$ |
| P_1 | P_2 | P_3 | P_4 |

The execution of the algorithm is traced in Figure 2. For each cycle we give the sequence of (processor, element) pairs in that cycle, the element determined to be the next in the merged list (i.e the output list), and the contents of the working stack. The algorithm is initiated by processor P_1 and we assume that write access is given to the processor with the smallest id# among those that want to talk. The termination of a cycle is detected by an empty time slot. Broadcasted elements are pushed into the work stack as they are heard. The Max element is popped from the work stack and joins the output list. The next cycle is then initiated by the top element.

```

main
begin
  /*initialization */
  CV <- the maximum value in list
  SUCC <- nil
  TERM <- nil
  BV <- nil

  repeat
    BV <- read next msg
    1. if BV is not empty then
      if CV > BV then broadcast CV
      if successful then call process_update
    2. if BV=empty then /*a cycle is terminated*/
      begin
        TERM <- terminator of the cycle
        if TERM = SUCC then
          broadcast & call process_update
        end
      3. if heard two empty slots then try to broadcast CV
        if successful then call process_update
  until list is empty

process_update
begin
  BV <- read next msg
  if BV = empty then
    begin
      remove CV from list
      if list is empty CV <- NIL else
        CV <- next element in list
      end
    else
      SUCC <- BV
  end
end

```

Figure 1: The merge algorithm

In the first cycle, 84 is determined to be the Max element. It is removed from the list of processor P_3 , the highest element of which then becomes 75. The second cycle is initiated by processor P_2 since it broadcast immediately before P_3 in cycle 1 and so on. The mechanism of a working stack suggests that rebroadcasting the element that initiates a cycle by a processor could be avoided altogether since the processors already heard that value and they can memorize it.

| | | | | | WORKING STACK | OUTPUT LIST |
|----------|----------------------|----------------------|----------------------|----------------------------------|---|--|
| cycle 1: | (P ₁ ,63) | (P ₂ ,79) | (P ₃ ,84) | <div>┌ └</div> output: 84 | <div>84 ← popped 79 ← init 63</div> | 84 |
| cycle 2: | (P ₂ ,79) | <div>┌ └</div> | | output: 79 | <div>79 ← popped 63 ← init</div> | 84 79 |
| cycle 3: | (P ₁ ,63) | (P ₂ ,64) | (P ₃ ,75) | <div>┌ └</div> output: 75 | <div>75 ← popped 64 ← init 63</div> | 84 79 75 |
| cycle 4: | (P ₂ ,64) | (P ₄ ,66) | <div>┌ └</div> | output: 66 | <div>66 ← popped 64 ← init 63</div> | 84 79 75 66 |
| cycle 5: | (P ₂ ,64) | (P ₄ ,65) | <div>┌ └</div> | output: 65 | <div>65 ← popped 64 ← init 63</div> | 84 79 75 66 65 |
| cycle 6: | (P ₂ ,64) | <div>┌ └</div> | | output: 64 | <div>64 ← popped 63 ← init</div> | 84 79 75 66 65 64 |
| cycle 7: | (P ₁ ,63) | <div>┌ └</div> | | output: 63 | <div>63 ← popped</div> | 84 79 75 66 65 64 63 |
| cycle 8: | (P ₁ ,54) | <div>┌ └</div> | | output: 54 | <div>54 ← popped</div> | 84 79 75 66 65 64 63 54 |

Figure 2. The execution of Merge algorithm on an example problem

Indeed, this is the basis of the improved Merge algorithm to be described later.

3.2 Correctness and complexity analysis

The correctness of the algorithm follows immediately from the following three facts:

1. The first element in each list is the largest in that list at all times.
2. In each cycle the maximum of all the first elements is determined.
3. The determined maximum is removed from its list and added to the output list.

In the complexity analysis we calculate only the number of broadcasts performed, since each broadcast is followed by a comparison stage. We show that the worst case complexity is $2n-1$. In order to prove that, we consider the following two lemmas.

Lemma 1:

Let CV_i denote the current value of processor P_i . Whenever CV_i is rebroadcast after the first time, it initiates a cycle.

Proof:

Assume to the contrary that the claim is not correct. There is, therefore, a situation in which a processor, that had already broadcast its current value, later hears a smaller value on the channel. The processor, in response, will rebroadcast its current value. Also, any current value that was broadcast must have a successor value and that successor is larger than itself. Consider the first cycle in which this situation occurs and let CV_i be the largest among the already-heard current values at the time of this cycle that hear a smaller value on the channel (case 1 in the algorithm). Let CV_j be the successor of CV_i at that time (any current value that was broadcast must have a successor). This successor is larger than CV_i and therefore it is also larger than the value on the channel. However, since we picked CV_i to be the largest one with this property we get a contrad-

iction.

□

It follows that from the time an element is first broadcast until it is merged, only values greater than or equal to itself can be broadcast. Let $\#V_i$ be the number of times element V_i is broadcast. The lemma implies that if $\#V_i > 1$ then V_i initiated at least $\#V_i - 1$ cycles. From lemma 1 we can conclude also that there is at most one predecessor to each current value. The reason is that a value is determined as a successor only during its first transmission (it cannot be the initiator of that cycle). Therefore it will be a successor to only one current value.

Lemma 2:

Each time V_i initiates a cycle (except for the last cycle which includes only V_i), the cycle is terminated by an element that has never been broadcast before.

Proof:

Assume $\#V_i > 2$. We arbitrarily choose the cycle which is initiated by the m^{th} broadcast of V_i ($1 < m < \#V_i$). Let the cycle be terminated by an element denoted u_i^m . If u_i^m participated in an earlier cycle, it must initiate all other cycles in which it participates (according to lemma 1), which leads to a contradiction.

□

This lemma implies that with each broadcast of V_i excluding the first and the last we can associate a distinct element which is broadcast just once. This element is removed immediately after it is broadcast. Since no two distinct elements initiate the same cycle, we can partition the set $\{V_1, \dots, V_n\}$ of all elements into disjoint subsets $M = \{S_1, S_2, \dots, S_r\}$ such that each subset S_i either consists of a single element, which is broadcast once or twice, or S_i consists of m elements, one of which, V_{j_i} , is broadcast $m-1$ times and the other $m-1$ elements are those which terminate each of the $m-1$ cycles initiated by V_{j_i} .

Thus:

$$1. \quad \forall i, j \quad S_i \cap S_j = \emptyset$$

2. if $|S_i| = m$ then the total number of broadcasts, $B(S_i)$ by elements in S_i , satisfies

$$B(S_i) \leq 2m$$

This leads to the following theorem.

Theorem 1:

Let $T(n, k)$ be the number of broadcasts performed by the Merge algorithm. $T(n, k)$ satisfies

$$n \leq T(n, k) \leq 2n-1 \quad (6)$$

Proof:

It is obvious that $n \leq T(n, k)$ since each element has to be broadcast at least once. Also

$$T(n, k) \leq \sum_{S_i \in M} B(S_i) \leq \sum_{S_i \in M} 2|S_i| = 2n \quad (7)$$

Since, the element which terminates the first cycle is broadcast just once. We are left with $n-1$ elements for which we have shown in (7) that the upper bound for their total broadcast time is $2(n-1)$, which yields:

$$T(n, k) \leq 2(n-1)+1 = 2n-1$$

□

3.3 An Improved Merge Algorithm

As mentioned earlier, it is possible to decrease the number of broadcasts required by making the initiation and termination of a cycle more sophisticated. However, these savings require a larger local memory for each processor.

In the modified version, each processor stores all the elements that were broadcast in a stack called *wstack* (as we did in the example). The initiation of cycles by elements that were broadcast before will be avoided altogether, since this information exists in the *wstack* of every processor. Each cycle now begins by broadcasting the second element relative to this cycle in the

original algorithm, and cycles with one element will now reduce to empty cycles.

At the beginning of a cycle each processor compares the value at the head of its list with the top element in the wstack and decides to broadcast only if its value is larger. The rest of the cycle proceeds in the same manner as in the previous algorithm where each processor pushes values onto wstack as it hears them. At the end of a cycle, (determined by an empty slot) each processor pops the top element from wstack. Any consecutive empty slot, following the first corresponds to an empty cycle (a cycle of 1 element in the previous algorithm). For each such empty slot a processor pops its wstack and this value joins the merged list. When wstack is empty but its list is not the processor knows that the initiation of a cycle by a value which was never broadcast before is called for.

The number of broadcasts required by this algorithm is exactly n . Each element is broadcast just once. The number of comparison stages, however, remains the same as before. Note that the new algorithm requires that each processor maintain a stack of size $O(k)$, thus presenting a tradeoff between the number of broadcasts and the size of local memory.

In the rest of this paper, whenever we talk about the "Merge by broadcast" algorithm we mean the first version unless otherwise specified.

3.4 Sorting algorithms

The Merge by broadcast algorithms can be used to sort n elements with k processors with IPABM by initially having each processor sort its own list, using some efficient sequential algorithm (such as quicksort or sequential Merge sort [1]). The merging phase is performed by our Merge-by-broadcast algorithm. Lets call this sorting algorithm Broad-Sort(n, k).

Theorem 3:

The complexity of Broad-Sort(n, k) is given by:

$$T(\text{Broad-Sort}(n, k)) = (\frac{n}{k} \log \frac{n}{k} + 2n - 1)t + (2n - 1)T = O(\frac{n}{k} \log \frac{n}{k} + n) \quad (8)$$

Proof:

Here we use the combined measure of performance that accounts for both comparison time and communication time.

The theorem is proved as followed:

$(\frac{n}{k} \log \frac{n}{k})$ - is the number of comparisons required to sort each list.

$(2n - 1)$ - is the number of comparisons for merging

$(2n - 1)$ - is the number of broadcasts

□

The maximum number of comparisons required for sorting a sequence of n elements on a sequential processor is asymptotically $n \log n$. Therefore, when k is smaller than $\log n$ the asymptotic speedup ratio of the the optimal sequential algorithm over the above algorithm is k , which is optimal. However, when k is greater than $\log n$, the total execution time required is asymptotically linear in n . Formally the speed-up ratio between sequential sorting and this Merge-Sort algorithm is greater then

$$\frac{k}{1 + \frac{k}{\log n}}$$

When the improved Merge algorithm is used to sort n elements with n processors ($k=n$) then we get the algorithm described by Levitan [15]. This algorithm uses exactly n broadcasts and $2n$ comparison stages. In this case (when $k=n$) the number of comparison stages can be reduced to n by having the elements which were not transmitted yet, keep a pointer to their relative order in the wstack. Each processor can do that with no extra cost since they listen to the channel anyway. In that case when a new cycle begins, with the top element in the wstack, the elements know their relation to it and they don't need to make the comparison.

This argument cannot be extended to the Merge-algorithm since the elements that are updated to be the new current values were not compared with the values that were broadcasted already.

Another way to sort n elements with n processors is as follows. Each processor broadcasts its value in a prespecified order. Each processor listens to the channel and remembers the value of its immediate successor in the list. After n broadcasts, all processors are linked in the order of their values. In the next phase the processors will broadcast their values again according to the order dictated by the linked list. The first will be the max value (the one with no successors). This algorithm uses $2n$ broadcasts and n comparison stages. Its advantage over Levitan's algorithm is that there is no contention on the channel, and therefore in the RPABM model (to be discussed later) no extra cost will have to be paid for accessing the channel. This algorithm cannot be extended to a Merge-algorithm (at least not in straight-forward way) without increasing the number of comparison stages significantly.

4. Optimality of the Merge-Algorithm

In this section we establish a lower bound on the performance of all "merge by broadcast" algorithms when the only criterion is the number of broadcasts performed. Consequently, the above Merge algorithms are shown to be optimal since they met this bound.

We consider all possible Merge algorithms using the IPABM to merge n distinct elements drawn from a set S on which an order is defined. The n elements are grouped into k sorted lists each with $\frac{n}{k}$ elements. There are k processors, each containing one of the sorted lists. The output is obtained in an independent processor (the output processor). We claim that any algorithm that merges the lists requires at least n broadcasts. More specifically it requires that each of the elements will be broadcast at least once.

This claim might seem trivial since for the output processor to create the merged list it must hear all the values! However if we reformulate the requirement such that the output processor need not know the actual values but simply their order, the claim is less obvious. Formally, let V_{ij} be the j^{th} element in processor P_i (or the j^{th} element in the i^{th} sorted list) and v_{ij} be the value of the element in this location for a given input of n elements. The output processor should be able to give, for each input, a series of locations $V_{i_1j_1}, V_{i_2j_2}, \dots, V_{i_kj_k}, \dots, V_{i_nj_n}$ such that the sequence of values $v_{i_1j_1}, v_{i_2j_2}, \dots, v_{i_kj_k}, \dots, v_{i_nj_n}$ is the final merged list. From what we will show it follows that the values themselves are also available at the output processor. First we prove our claim for the special case of n lists having 1 element each.

Lemma 3:

To merge n lists of 1 element each, with IPABM using n processors, each of the elements must be broadcast.

Proof (sketch):

Any two elements which are adjacent in the sorted list must be compared directly. Let a_i, a_{i+1} be two consecutive elements. The order between these two elements and the rest is exactly the same, thus, to determine their internal order they must be compared directly. Since a_i and a_{i+1} are located in different processors and the outcome of the comparison must be available at the output processor which doesn't know them initially, both values must be broadcast. Each one of the processors including the output processor can then compare and determine the order between the two elements. It might be argued that it is sufficient to have one processor broadcast its value and the other only indicate whether it is larger or smaller, however, we count all messages in the same way and since the broadcast of the value gives more information we assume the values themselves are transmitted. We can conclude that since there are $n-1$ adjacent pairs, $n-1$ comparisons are required, each corresponding to two broadcasts. Since $n-2$ of the elements participate in two adjacent pairs the number of broadcasts required is at least

$$2(n-1) - (n-2) = n \quad (9)$$

□

In the general case, each processor has $\frac{n}{k}$ sorted elements. As argued earlier, any adjacent pair of elements must be compared. If two adjacent elements are in the same processor it is not necessary to broadcast both elements in order to make a comparison (a processor can make the comparison and then just broadcast the result in some coding or broadcast only the larger value). However, it is possible to create an input for which no two adjacent elements are located in the same processor. Figure 3 illustrates such a case. Let $a_1 > a_2, \dots, a_n$ be the sorted list. The list of elements in processor P_i ($1 \leq i \leq k$) is $a_1, a_{1+\frac{n}{k}}, \dots, a_{i \cdot \frac{n}{k}}$.

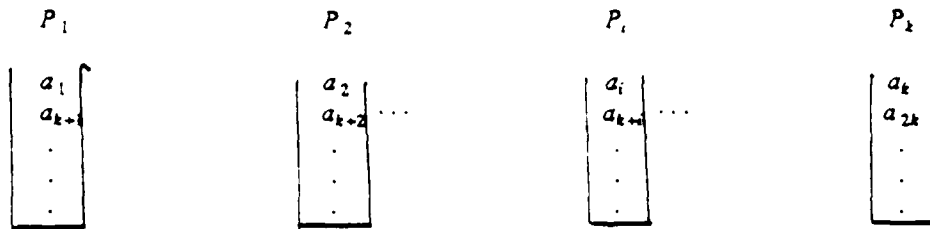


Figure 3: A worst case example

Thus, any comparison between adjacent values requires that both of them must be broadcast. This yields the following theorem.

Theorem 2:

Any Merge-by-broadcast algorithm, using IPABM, with n elements and k lists (k processors) requires n broadcasts in the worst case if the output is accumulated in a separate output processor.

□

It is easy to show that when the output processor is a processor which contains one of the lists the lower bound decreases to $n - \frac{n}{k}$ broadcasts (the output processor need not broadcast its own values).

From Theorems 1 and 2 we conclude that the Merge by broadcast algorithm presented in Section 3 is asymptotically optimal w.r.t. number of broadcasts, while its modified version is absolutely optimal.

5. Access Scheme Considerations

So far in our discussion we have assumed the existence of an "ideal" access scheme that resolves all conflicts in constant time. Even if such an access scheme is not available, this assumption is appropriate when the algorithm itself is designed so that conflicts never arise (i.e., in each time slot at most one processor is enabled) as in the last sorting algorithm in section 3.4. If the algorithm is not designed in this way, conflicts between processors will generally arise and the access scheme used may have a profound effect on the complexity of the algorithm.

Numerous access schemes have been proposed and analyzed [6, 10, 12, 13, 11, 22] in the context of broadcast communications. The measures that are used for evaluating their performance are those of channel capacity, throughput and delay. Of most interest to us is capacity; in particular, we are interested in the ratio between the time the channel is used for conflict resolution and the time it is used for "useful" communication.

We now modify our ideal model to include these access considerations. Let RPABM be an IPABM with the following changes: the channel is slotted into two types of time slots: one of length T for message transmission and the other one of length τ . The RPABM has a Conflict Resolution Protocol, CRP, which substitutes for the global access mechanism in IPABM. The CRP is invoked whenever a conflict arises and it uses a τ -slotted channel. A T slot carrying a transmission marks the termination of the CRP. Usually $\tau \ll T$.

The communication time of an algorithm with RPABM now includes the number of T slots required for real transmissions and the number of τ slots used by the CRP. Let $\#T$, $\#\tau$, be the maximum number of T slots and the maximum number of τ slots respectively, used by algorithm

A for broadcasts and for resolving conflicts given a specific CRP. The worst case communication time for algorithm A with CRP, denoted by $Comm(A,CRP)$ is defined as:

$$Comm(A,CRP) = \#T \cdot T + \# \tau \cdot \tau \quad (10)$$

Since $\tau \ll T$, the contribution of the second term, (the cost of accessing), might be negligible for some specific parameters of the problem. However, for the asymptotic complexity, τ cannot be ignored. The ratio $\frac{\# \tau}{\# T}$ characterizes the impact of the access scheme used on the asymptotic complexity of the algorithm. Specifically, if $\frac{\# \tau}{\# T} = O(f(k))$, where $f(k)$ is a nondecreasing function of k (the number of processors), then it is easy to see that

$$Comm(A,CRP) = O(\#T \cdot (f(k) + 1)) \quad (11)$$

In the next two subsections we present the "Merge by broadcast" algorithm using two realistic access schemes: MSAP (mini-slot alternating priority) [11] and the Tree-Algorithm [6]. Following that we give some lower bounds for CRP performance.

5.1 Merge with MSAP

In the MSAP scheme, processors obtain permission to broadcast according to some predetermined priority order among them dictated, for example, by their id's. Processors may broadcast one after the other in a round robin fashion, and when a processor has nothing to say its turn is passed to the next in order after an empty τ -slot. When it does broadcast it uses a T slot, and then gives the turn to the next processor. This is similar to a token ring.

This method is very appealing for the Merge-algorithm since we have cycles built into it in which each processor might want to transmit once. Thus, integrating the access scheme into the Merge-algorithm is straightforward: processors are ordered in increasing order of their id's. Any cycle of the algorithm is initiated by its initiator if the algorithm determines one. Otherwise P_1 is the initiator. If P_1 is an initiator of a cycle then the next one that can talk is P_{1+1} , and then P_{1+2} and so on until the end of the cycle. A processor determines when its turn comes by detecting

the end of a cycle, by knowing the initiator's id and by counting the number of τ and T slots that occurred.

In Figure 4 we show how the algorithm works with MSAP using the same input as in Figure 2. Note that we do not need an extra time slot to determine the end of a cycle with this CRP. The empty slots are of length τ and are noted by " ", the full slots are of length T .























| | | | | | output: | |
|----------|----------------------|---|---|---|---|----|
| cycle 1: | (P ₁ ,63) | (P ₂ ,79) | (P ₃ ,84) |  |  | 84 |
| cycle 2: | (P ₂ ,79) |  |  | |  | 79 |
| cycle 3: | (P ₁ ,63) | (P ₂ ,64) | (P ₃ ,75) |  |  | 75 |
| cycle 4: | (P ₂ ,64) |  | (P ₄ ,66) | |  | 66 |
| cycle 5: | (P ₂ ,64) |  | (P ₄ ,65) | |  | 65 |
| cycle 6: | (P ₂ ,64) |  |  | |  | 64 |
| cycle 7: | (P ₁ ,63) |  |  |  |  | 63 |
| cycle 8: | (P ₁ ,54) |  |  |  |  | 54 |

Figure 4: Example of Merge with MSAP

We next determine the communication complexity of the Merge algorithm with the MSAP CRP.

Theorem 4:

For any instance I of Merge(n, k) with RPABM when CRP is MSAP, the number of τ slots used, $\# \tau$, satisfies

$$\frac{n \cdot (k-1)}{2} \leq \# \tau \leq n \cdot (k-1) \quad (12)$$

Proof:

Since the number of cycles is n and in each cycle we can have at most $k-1$ empty τ -slots (when there is just one transmission for example) there are at most $n \cdot (k-1)$ τ slots i.e.:

$$\# \tau \leq n \cdot (k-1)$$

We now determine the lower bound. An element is added to the Merged list when it terminates a cycle. To determine if a cycle is terminated by an element of processor P_i , $k-i$ empty τ -slots must pass by. Since there are $\frac{n}{k}$ elements in processor i and each terminates a cycle once we have

$$\# \tau \geq \frac{n}{k} \sum_{i=1}^k (k-i) = \frac{n}{k} \sum_{i=0}^{k-1} i = \frac{n \cdot (k-1)}{2} \quad (13)$$

□

We can conclude that using MSAP we have $\# \tau = \Theta(n \cdot k)$. Altogether

$$Comm(Merge(n, k), MSAP) = (2n-1) \cdot T + (n \cdot k) \cdot \tau = \Theta(n \cdot k) \quad (14)$$

Note that

$$\frac{\# \tau}{\# T} = O(k) \quad (15)$$

Thus the asymptotic complexity increased by a factor of k which is substantial when k is not a constant but rather a function of n .

5.2 Merge with the Tree-Algorithm

To apply the MSAP access scheme to the Merge-algorithm we took advantage of the structure of the algorithm and its decomposition into cycles. The access scheme resolves conflicts for each cycle of transmissions and not for each transmission independently.

Our approach in the following scheme is to consider each conflict independently without acquiring information from previous conflicts or previous steps of the algorithm. Whenever a conflict occurs, the CRP is invoked and gives the right to transmit to one of the involved proces-

sors. The question we want to address is: Given k processors that want to transmit, how many time slots (for conflict resolution) are needed until one of the processors succeed to broadcast (this is essentially the well known election problem in distributed computing). Greenberg [10] addresses similiar questions however none of them is identical to our question and therefore none of his results are the same. For instance he considers the problem of having k processors that want to talk in the same slot and provides a probabilistic protocol which on the average enables all the k messages to be posted in time $O(k)$. In our case, however, the situation changes after each successful broadcast, namely, if a processor wanted to talk and another processor was chosen to broadcast, it may not want to talk after it heard the broadcast message.

We apply Capetanakis' tree algorithm [6] to resolve conflicts using, again, the processor id's. The Tree-CRP is described by the recursive procedure $\text{Tree}(r,j)$ in Figure 5. Whenever a processor wants to transmit, the procedure $\text{Tree}(1,k)$ is invoked, where k is the number of processors.

$\text{Tree}(r,j)$ /*executed by processor P_i */

while you want to talk Do

 broadcast in the next time slot

 if no collision then broadcast and end

 else if $r \leq i \leq \left\lfloor \frac{r+j}{2} \right\rfloor$ then $\text{Tree} \left(r, \left\lfloor \frac{r+j}{2} \right\rfloor \right)$

 else if the next τ -slot is empty then $\text{Tree} \left(\left\lfloor \frac{r+j}{2} \right\rfloor, j \right)$

endwhile

Figure 5: The Tree-CRP

All processors try to broadcast in the first slot. If there is a collision, only those with id# less than $\frac{k}{2}$ try to transmit again. Another collision enables only those processors with id# $\leq \frac{k}{4}$ to transmit and so on until a successful transmission or an empty slot occurs. The latter event activates another subset of processors to keep trying in the same manner. The CRP uses a

sequence of τ -slots which are either collision slots or empty slots, and terminates by a successful transmission, i.e. by a T slot.

Resolving one conflict using the Tree-Algorithm may require $2 \cdot \log k$ slots in the worst case. The scheme can be easily modified to take only $\log k$ τ - slots by noticing that an empty slot implies a collision in the subsequent slot and can therefore be skipped.

Theorem 5:

The number of τ -slots, required by the Merge(n,k) algorithm, in its two versions, with RPABM when the Tree CRP is used, satisfies $\# \tau \leq n \cdot \log k$.

Proof:

The number of broadcasts in the improved algorithm is n . In the first version of the algorithm there are $2n$ broadcasts but only the first broadcast of each value may be involved in conflicts on the channel. In subsequent broadcasts the value initiates a cycle in which case it is the only one to access the channel. Therefore, for both versions of the Merge algorithm there are n broadcasts that may be involved in a conflict. Since $\log k$ τ -slots are used for solving the conflict, the claim follows.

□

We conclude that using the Tree-CRP and the first version of the Merge algorithm the communication time is:

$$Comm(Merg(n,k), Tree-CRP) = (2n-1) \cdot T + n \cdot \log k \cdot \tau = O(n \log k) \quad (16)$$

In this case

$$\frac{\# \tau}{\# T} = O(\log k) \quad (17)$$

We now show that any algorithm for conflict resolution can do no better than the Tree-CRP. First, we introduce some formalism. Let a Conflict-Resolution-Protocol, CRP, for a set of k processors $\{1, \dots, k\}$ be a function from the power set of $\{1, \dots, k\}$ to the set $\{1, \dots, k\}$. The CRP

determines for any subset of conflicting processors, one processor that can talk.

The execution of CRP for any subset of processors is over the τ -slotted channel where each slot is either an empty slot or a conflict slot. The last slot is of length T . The sequence of empty slots and conflict slots up to but not including the T slot could be considered the encoded information by which CRP selects a specific processor. Note that a processor does not know which subset is currently being worked on by the CRP but only whether or not it belongs to this subset. Let $C(S,x)$ be the binary code (i.e. the sequence of empty and conflict slots) which result when all processors in a subset of processors S want to transmit and x is the first who succeeds. Two properties are required from any CRP.

1. If $CRP(S)=x$ where S is a subset of $\{1,2,\dots,k\}$ then $x \in S$.

2. If S_1, S_2 are any two subsets such that $x \in S_1 \cap S_2$ and if

$$CRP(S_1) = x$$

and

$$CRP(S_2) = y \neq x$$

then

$$C(S_1, x) \neq C(S_2, y)$$

Let $l(CRP)$ be the maximum code length of a CRP.

Theorem 6:

For every CRP defined over a set of k elements $l(CRP) \geq \log k$.

Proof:

For any given CRP we create a special family of subsets of processors

$$CORE(CRP) = \{S_1, S_2, \dots, S_k\}$$

in the following recursive way:

$$S_1 = \{1, \dots, k\}$$

$$S_{i+1} = S_i - \{x_i\}$$

where

$$x_i = CRP(S_i)$$

Obviously,

$$S_k \subset S_{k-1} \subset \dots \subset S_3 \subset S_2 \subset S_1,$$

also,

$$CRP(S_i) \neq CRP(S_j)$$

We now show that the code sequences for $CRP(S_i)$ are all different. That is

$$\forall i, j \quad C(S_i, x_i) \neq C(S_j, x_j)$$

Suppose this is not true and for some i, j

$$C(S_i, x_i) = C(S_j, x_j)$$

where $j > i$. Then

$$S_i \supset S_j \Rightarrow x_j \in S_i$$

Thus, $x_j \in S_i \cap S_j$ with $CRP(S_j) = x_j$ and $CRP(S_i) = x_i \neq x_j$, but $C(S_i, x_i) = C(S_j, x_j)$ which contradicts property 2 of the CRP. This proves that every CRP must create at least k different binary codes which is known to require $\log k$ binary slots.

□

We see that any deterministic CRP does add a complexity factor of $\log k$ for every broadcast that enables more than one processor. It can be shown that even when we limit the size of conflicts to only two out of the k processors the worst case complexity of any CRP is still $\log k$. The argument goes as follows: In case of a conflict a CRP assigns a subset of the processors to talk in the first time slot and the other to be silent. If there is exactly one processor talking, the conflict is resolved and the CRP stops. Otherwise, in case of a collision, a subset of the colliding processors can be chosen for the second time slot. If there was silence in the first slot, a subset of the remaining processors will be selected for the second time slot etc. In the worst case the two

colliding processors can always be in the subset which is larger in each step of division, therefore, only after $\log k$ steps (there are k processors) the CRP will stop. This argument provides a different proof to theorem 6 as well. This suggests approaching the design of broadcast algorithms in a way that minimizes the number of broadcasts that can result in conflict or not to allow conflicts at all.

Conclusion

In this paper we addressed issues of design and complexity involved in incorporating "broadcast communication" into distributed algorithms. We presented algorithms for merging k lists of $\frac{n}{k}$ elements each by k processors and proved the complexity to be $O(n)$, regardless of the number of lists (processors). We also showed that this performance is optimal under the scheme of one-channel broadcast.

We initially avoided the effect of conflicts which exist in this mode of communication by introducing the algorithm in an ideal environment in which we pay no penalty for accessing the channel. We then showed that the problem of accessing the channel adds a factor of at least $\log k$ to the algorithm's performance. This suggests a need to investigate whether a different approach; i.e., minimizing the number of conflicts while designing the algorithm, might result in a better total performance. We also note that the use of a single channel limits the performance considerably (for example, merging cannot be accomplished in less than n sequential time slots) which motivates the use of more complex configurations of broadcast with more than one channel. For recent work on broadcast networks with multiple channels see [14, 17].

Acknowledgments

We would like to thank Eli Gafni for many stimulating discussions and helpful remarks, and John Marberg for many comments and a thorough review of the manuscript.

References

- [1] Aho, Hoipcroft, and Ullman, *The Design and Analysis of Computer Algorithms*, Reading, Massachusetts: Addison Wesley, 1974.
- [2] Batcher, K., "Sorting networks and their applications," in *Proceedings AFIPS Spring Joint Comp. Conf.*, 32, 1968, pp. 307-314.
- [3] Baudet, Gerard and David Stevenson, "Optimal sorting algorithms for parallel Computers," *IEEE Transaction on Computers*, Vol. C-27, No. 1, 1978.
- [4] Bokhari, Shahid H., "Max: An algorithm for finding maximum," in *Proceedings The 1981 Conference on parallel processing*, 1981, pp. 302-303.
- [5] Borodin, A. and J.H. Hopcroft, "Routing, merging, and sorting in parallel models of Computations," in *Proceedings 14th ACM Symposium on Theory of Computing*, 1982.
- [6] Capetanakis, J., "Generalized TDMA: The multiaccessing tree protocol," *IEEE Transactions on Communications*, Vol. COM-27, No. 10, 1979, pp. 1476-1484.
- [7] Cook, S. and C. Dwark, "Bounds on the time for parallel RAM's to compute simple functions," *14th ACM Symposuim on Theory of Computing*, 1982.
- [8] Dechter, Rina and Leonard Kleinrock, "Parallel algorithms for multiprocessors using broadcast channel," UCLA, ATS, CSD, Los Angeles, California, Tech. Rep. 81002, 1981.
- [9] Ditton, Dina, David J. Dewitt, David K. Hsiao, and Jaishankar Menon, "A taxonomy of parallel sorting," *Computing Surveys*, Vol. 16, No. 3, 1984, pp. 287-318.
- [10] Greenberg, A.G., "On the time complexity of broadcast communication schemes," in *Proceedings 14th ACM Symposium on Theory of Computing*, 1982.
- [11] Kleinrock, and M. O. Scholl, "Packet Switching in radio channels: New conflict-free multiple access schemes for a small number of data users," *IEEE Transaction on Communications*, Vol. COM-28, 1980, pp. 1015-1029.
- [12] Kleinrock, L., *Queuing Systems*, Vol 2., New York: Jon Wiley, 1976.